Programming Methodology Group, Department of Computer Science, ETH Zurich

A Linear Type System for Gobra

Prepared by: Han Liming
Legi-Nr: 20-960-100
Supervisors: Felix Wolf, João Pereira, Prof. Dr. Peter Müller
Period: March 2023 - September 2023

Abstract

Gobra is a verification tool for Go programs and it utilizes permission reasoning to control the usage of heap-allocated values. Nevertheless, this approach may lead to an annotation overhead and prove to be time-consuming. A linear type system can efficiently improve program reliability and resource management by enforcing strict rules on resource usage.
In this thesis, we design a linear type system for Gobra to simplify memory reasoning. We introduce some new features to extend the current type system and integrate linearity into Gobra. By separating memory reasoning from the program's verification, the verification time is reduced. At the same time, by analyzing some examples, the annotation workload of the programmers also decrease significantly, demonstrating the practical usefulness of the linear type system.

Acknowledgments

I am deeply grateful to Felix Wolf and João Pereira for their invaluable guidance and support throughout this project. My understanding of the type system was very limited at the beginning of the project. With their guidance, I gradually found the right direction to complete the project. Not only did they provide helpful suggestions, but they also keenly pointed out the potential problems in the system to refine the design. I also benefited greatly from Felix and João's patience and unstinting knowledge sharing. They would patiently answer my questions and sometimes even provide notes on important knowledge points in the meeting to ensure that I could fully understand.
I would also like to express my gratitude to Prof. Peter Müller and every member of the Programming Methodology Group. Their feedback and thought-provoking questions at different stages of the project have helped me to think more deeply about the design, and have inspired me to better improve the project. I feel very lucky to have worked in such an outstanding team, and I am profoundly inspired and encouraged by these exceptional supervisors. Thank you once again for your support!

Contents

Chapter 1

Introduction

Go is a modern open-source programming language designed for efficiency, simplicity, and scalability. Go was developed to address the challenges of contemporary software development, providing a balance between expressive syntax and high performance. Go boasts features such as strong typing, garbage collection, and concurrency support through its goroutines and channels, making it particularly suitable for building concurrent and distributed systems. Go's widespread adoption in various domains, including web development, system programming, and cloud computing, underscores its versatility and relevance in the modern programming landscape. Meanwhile, Go provides an extensive suite of testing tools that aid developers in finding issues in their programs. However, these testing tools cannot guarantee the absence of bugs. To address this limitation, Gobra [1], a program verifier for Go, was developed.

Gobra is a verifier for Go programs. Gobra takes Go programs annotated with specifications and proof annotations, and checks whether the program satisfies the given specification. If it does not, Gobra provides additional information that may be useful for debugging purposes. To specify and verify programs that deal with heap-allocated data structures, Gobra uses the ideas from *Separation Logic* [2], where every memory location on the heap is associated with a permission. If a method wants to access a memory location, it is required to hold the related permission [1]. However, permission reasoning is expensive. In particular, permission reasoning introduces an annotation overhead and increases the verification time. As detailed in the work by Wolf et al. [1], if more memory locations that are tracked with permissions are used, then Gobra's verification time will increase because the verifier has more information to consider. In the worst case, verification may be unsuccessful due to the non-termination of the verification process.

A linear type system is a type system that introduces a new type modality called *linear type* to help manage the usage and deallocation of locations [3]. In a traditional type system, values can be used multiple times, whereas a linear type system enforces that linear values can only be used once to guarantee memory safety. Linear type systems have been widely studied and used in programming languages, especially in areas that require fine-grained control of resource management.

As previously mentioned, using permission reasoning in Gobra not only increases the burden on programmers for annotating but also increases the overhead of verification. The efficiency of linear type systems in simplifying memory reasoning has been illustrated by Rust verifiers, such as Creusot [4] and Prusti [5]. Additionally, Linear Dafny [6] also demonstrates that incorporating linear types can simplify the proof obligations ultimately passed to the backend, and achieve better verification performance. According to Linear Dafny [6], the linearized VeriBetrKV has 28% fewer lines of proofs, and 30% shorter verification time overall. Thus, inspired by these papers, our intention is to design a linear type system for Gobra to reduce the annotation overhead and verification time when verifying the memory safety of programs with linear access to memory.

1.1 Challenges

Designing a linear type system for the Gobra presents several challenges due to the language's characteristics and design philosophy. Due to varying design requirements, there exist different variants of linear type systems. For instance, some linear type systems do not permit linear values to be shared, while others allow for the safe sharing of linear values. Rust [7] achieves safe sharing by introducing the concept of *borrowing*, which will be will be described in detail in Section 3.1. In this type of system, it is crucial to ensure the safety of all references to a specific location. We have a large design space within which we need to make careful trade-offs. Finding the design that best suits the Go programming language is the first challenge.

The second challenge comes from the desire to support as many language features as possible. Go has a rich set of features and diverse data structures, in which case the complexity of designing a linear type system for Gobra significantly increases. It is a great challenge to come up with a unified design that can accommodate different data structures including the built-in types of Go like arrays, slices, and maps.

The third challenge pertains to maintaining compatibility with the existing Gobra system. Ideally, the introduction of a linear type system in Gobra should not invalidate any existing Gobra codebases. While the incorporation of a linear type system may lead to increased efficiency and simplicity, it is important to carefully consider compatibility, performance, and the system's overall impact on Gobra's existing ecosystem.

In summary, the complexity inherent in designing a linear type system stems from the need to harmonize with the nature of the Go language, cover a wide range of features and data structures, and maintain compatibility with the pre-existing Gobra codebases. All these challenges highlight the complexity and multifaceted nature of designing a linear type system.

1.2 Outline

In this paper, we develop a linear type system to complement permission reasoning for managing heap-allocated values in Gobra.

Chapter 2 offers a detailed analysis of the Gobra system, providing the essential background knowledge to understand the subsequent parts of the thesis. In particular, this chapter provides an in-depth introduction to the permission reasoning mechanism in Gobra, which is crucial to comprehend our novel design of the linear type system.

Chapter 3 provides some background information related to linear type systems, which is the foundation for explaining our design in the following chapters.

Chapter 4 presents our design of the linear type system for Gobra, starting from an overview of the design and subsequently breaking down the design into modular components.

Chapter 5 presents a series of use cases for analysis, demonstrating the features supported by our linear type system.

Chapter 6 concludes the thesis, and outlines potential avenues for future improvements and research directions.

Chapter 2

Gobra

This chapter details the main features of Gobra related to our work which are necessary to understand this thesis. In Section 2.1, we briefly introduce the annotation mechanism in Gobra. Section 2.2 delves into an in-depth explanation of how Gobra manages the permissions of heap-allocated values. More insights into Gobra and its functionalities are available in the publication by Wolf et al. [1]. Furthermore, a comprehensive tutorial on Gobra is available on Github [8].

2.1 Annotations

As previously mentioned, Gobra takes Go programs annotated with formal specifications as input and outputs whether the given Go project satisfies all specified requirements. Programmers are required to annotate the Go programs to constrain the desired behavior. Within the Gobra system, the supported annotations include *assertions*, *preconditions*, *postconditions*, and *loop invariants*.

An *assertion* is the most basic annotation supported in Gobra, which is used to check a specific logical statement's validity at a given program point. Figure 2.1 is an example using assertion. In line 1, we define `x` as an integer variable and assign `1` to `x`. In line 2, `x` remains unchanged and equals to 1, resulting in the successful verification of the assertion `assert x == 1`. In an assertion, Gobra supports to use of Go's side-effect-free and deterministic boolean expressions (e.g., `x == y`), implications (`==>`), conditionals (`cond ? e1 : e2`), conjunctions (`&&`), universal quantifiers (e.g., `forall i int :: i < 10 ==> i < 5`), and existential quantifiers (e.g., `exists x int :: x > 42`).

```
x := 1
assert x == 1
```

Figure 2.1: Example code of using assertions in Gobra

Preconditions and *postconditions* are used to constrain the behavior of functions in programs. *Preconditions* are mainly used to guarantee the specifications hold when the function is invoked. Thus, the caller is responsible for ensuring that these preconditions are met. Preconditions are usually the constraints on the parameters of the function. In contrast, *postconditions* outline requirements that the final state needs to satisfy. In general, if a method accesses a memory location, it is required to hold the associated permission. Permissions are transferred between methods upon preconditions and postconditions, and permission reasoning will be described in detail in the next section. Figure 2.2 shows an example using function specifications for the function `add5`, which adds `5` to the argument passed in. Line 1 is a precondition, and this line requires the parameter `n` to be non-negative. Line 2 is a postcondition that constrains the result of this function to be `5` more than the argument.

```
1  requires 0 <= n // precondition
2  ensures  res == n + 5 // postcondition
3  func add5(n int) (res int) {
4    res := n + 5
5    return res
6  }
```

Figure 2.2: Example code of using function specifications in Gobra

Loop invariants are used to describe invariant properties in a loop procedure. Loop invariants have to hold before and at the end of each loop iteration [8]. Figure 2.3 is an example using loop invariants. Line 2 and line 3 are invariants used to verify the loop, which guarantee that before and after each iteration of the loop, `i` ranges from 0 to 10 and `sum` equals to `i * (i-1)/2`.

```
1  assert sum == 0
2  invariant 0 <= i && i <= 10
3  invariant sum == i * (i-1)/2
4  for i := 0; i <= 10; i++ {
5      sum += i
6  }
7  assert sum == 45
```

Figure 2.3: Example code of using loop invariants in Gobra

2.2 Alias and Permission Reasoning

This section presents the permission reasoning mechanism in Gobra and a brief overview of the existing type system in Gobra, which is helpful for understanding the linear type system we designed. Section 2.2.1 introduces the reason why we need permission reasoning in Gobra. Section 2.2.2 illustrates how Gobra manages permissions to memory locations. Lastly, Section 2.2.3 details two types of memory locations in Gobra's type system.

2.2.1 Reference and Alias

References and aliases allow multiple variables to interact with the value in the same memory location. A reference points to a location, allowing variables to indirectly access and modify the value stored in the location. This indirection enables variables to share data and synchronize changes. A memory location is aliased if two or more variables hold references to this location. In object-oriented programming, the use of references is intended to offer efficiency and data-sharing benefits. However, improper handling of references can lead to issues.

Figure 2.4 is an example illustrating the problem of aliasing caused by destructive updates. In this Go example, we define a `Student` struct and create an instance, `studentA`. Then, we create two references `studentB` and `studentC` that point to the same memory location of `studentA`. Modifications to `studentC` change the underlying value of `studentA` and also inadvertently affect the print result of `studentB`, leading to unexpected changes. This highlights the potential pitfalls of using references without proper care.

```go
type Student struct {
    name string
    age  int
}

func main() {
    studentA := Student{"Alice", 20}

    studentB := &studentA
    studentC := &studentA

    fmt.Println(studentB.name)  // Output: "Alice"
    fmt.Println(studentB.age)   // Output: 20

    studentC.name = "Bob"
    studentC.age = 22

    fmt.Println(studentB.name)  // Output: "Bob", the output changed
    fmt.Println(studentB.age)   // Output: 22, the output changed
}
```

Figure 2.4: Example code of aliasing

Therefore, managing references and aliases is crucial for maintaining code clarity, avoiding bugs, and ensuring proper memory management. In Gobra, the management of references is facilitated by permission reasoning mechanisms.

2.2.2 Permission Reasoning in Gobra

In Gobra, reasoning about heap-allocated values is based on a variant of *Separation Logic* [2], Implicit Dynamic Frames. A permission is associated with a heap location. Permissions are acquired by method executions and are exchanged between methods using preconditions and postconditions. The permission to modify a location `v` is denoted in Gobra by `acc(&v)`.

Due to the non-destructive nature of read permission, references for reading data from the same memory location can coexist. However, references for writing data cannot coexist with any other references. In Gobra, we use fractional permission to guarantee this, which stipulates that each reference possesses a certain amount of permission (greater than 0 and less than or equal to 1). Additionally, the sum of the permission amounts for references pointing to the same location cannot exceed 1. In Gobra, only the reference with full (exclusive) permission, denoted as `acc(&v)` or `acc(&v, 1/1)`, is allowed to modify the related location. And `acc(&v, i) (0<i<1)` means read-only permission to location `v`. We can also use _ to represent a non-zero permission amount, denoted as `acc(&v, _)`.

Figure 2.5 is an example of permission reasoning. The function `addTwoSlice` adds the corresponding elements of two slices, `s1` and `s2`, with equal lengths and assigns the result to `s1`. Therefore, the function `addTwoSlice` needs full permission of `s1` and read permission of `s2`. Line 2 and line 3 obtain the write permission of `s1` and the read permission of `s2`, respectively. The permissions are returned back by line 4 and line 5.

It's worth mentioning that the permissions to heap-allocated values accessed within a loop body are lost, unless they are encompassed within an invariant, as shown in line 9 and line 10. We use *old expression* to refer to the old value of expression `e` in the state of the precondition, indicated by `old(e)`. As shown in lines 6 and 7, the postcondition `forall k int :: 0 <= k && k < len(s1) ==> s1[k] == old(s1[k]) + old(s2[k])` constrains that after calling the function `addTwoSlice`, the value of `s1[k]` equals to the sum of `s1[k]` + `s2[k]` when the function was invoked.

```
requires len(s1) == len(s2)
requires forall k int :: 0 <= k && k < len(s1) ==> acc(&s1[k])
requires forall k int :: 0 <= k && k < len(s2) ==> acc(&s2[k], 1/2)
ensures  forall k int :: 0 <= k && k < len(s1) ==> acc(&s1[k])
ensures  forall k int :: 0 <= k && k < len(s2) ==> acc(&s2[k], 1/2)
ensures  forall k int :: 0 <= k && k < len(s1) ==> s1[k] == old(s1[k])
         + old(s2[k])
func addTwoSlice(s1 []int, s2 []int) {
  invariant 0 <= i && i <= len(s1)
  invariant forall k int :: 0 <= k && k < len(s1) ==> acc(&s1[k])
  invariant forall k int :: 0 <= k && k < len(s2) ==> acc(&s2[k], 1/2)
  invariant forall k int :: i <= k && k < len(s1) ==> s1[k] == old(s1[k])
  invariant forall k int :: 0 <= k && k < i ==> s1[k] == old(s1[k]) +
            old(s2[k])
  for i := 0; i < len(s1); i += 1 {
    s1[i] = s1[i] + s2[i]
  }
}
```

Figure 2.5: Example code of permission reasoning in Gobra

2.2.3 Shared and Exclusive Memory Locations

In Gobra, values are categorized as either *shared* or *exclusive*. *Exclusive* values are only available through a single method execution and can be treated as stack-allocated local variables, which behave like mathematical values [1]. Thus, exclusive values will never alias. Conversely, *shared* values are situated on the heap and are accessible by multiple methods and threads [1], which means shared values may alias. Section 2.2.1 underscores the significance of managing references. In Gobra, permission reasoning is applied to shared locations because of the possibility of aliasing. This is crucial to ensure the absence of data race conditions and the safe modification of memory. Thus, Gobra requires shared values to be annotated with an additional `@` symbol at the point of declaration [1].

Figure 2.6 is a sample code having shared and exclusive variables. Line 1 declares an exclusive int variable `n`, and line 2 declares a shared array `a` of fixed length 4. The function `incr` increases all elements in the given slice (the first argument) by `n` (the second argument). The array `a` is sliced in line 3, in which case we actually create a reference to `a`. Thus, in Gobra, the programmer is required to add `@` modifier when declaring `a`, as shown in line 2. Omitting the `@` annotation will cause a type error.

```
1 n := 2
2 a@ := [4]int { 1, 2, 4, 8 }
3 incr(a[2:], n)
```

Figure 2.6: Example code of the shared and exclusive variables in Gobra

2.3 The Judgments for Gobra's Type System

Expression judgments and *statement judgments* are basic components of a type system. An *expression judgment* determines the type of an expression. An expression is a combination of variables, constants, and operators that produce a value. An expression judgment validates whether the expression adheres to the language's type rules, ensures that the expression is well-formed, and assigns a specific type to the resulting value. *Statement judgments* are used to check the validity of statements. Statements are commands that perform some operations but do not produce values, such as variable declarations, assignments, and so on. Statement judgments ensure that statements conform to the language's syntactic and semantic rules.

To understand the typing rules in the following subsections, we list all symbols that are used afterwards in Table 2.1. $\circ$ is the exclusive modifier and @ is the shared modifier. The symbol M is the modifier of a location, which could potentially be shared or linear. S represents the type context, which maps variables to their types (T) and modifiers (M). Lastly, s represents a statement.

Symbol	Meaning
$\circ$	Exclusive Modifier
@	Shared Modifier
M	Modifier, $M := \ \circ \mid @$
T	Type
S	Type Context, $S : \ \mathit{variable} \mapsto T\ M$
e	Expression
x	Variable
s	statement

Table 2.1: Notation used in typing judgments - Gobra

2.3.1 Expression Judgment

Gobra's expression judgment of form $S \vdash e : \ T\ M$ asserts that the expression e has type T and modifier M in the given context S. The following are the main expression judgments of Gobra's old type system.

$$\text{Variable} : \frac{}{S \vdash x : \ T\ M} \tag{2.1}$$

This is the most basic typing rule named `Variable` (2.1), and we will get the type and modifier of the given expression e by applying this rule.

$$\text{Dereference}: \frac{S \vdash e: \ *T^{\circ}}{S \vdash *e: \ T@} \tag{2.2}$$

This is `Dereference` rule (2.2). From the premise (above the line), it is evident that e is a reference to a value of type T. Thus, if we dereference e, we will obtain a value of type T with the modifier @. This is due to a constraint in Gobra that the value being referenced must be shared.

$$\text{Reference}: \frac{S \vdash e: \ T@}{S \vdash \&e: \ *T^{\circ}} \tag{2.3}$$

This is `Reference` rule (2.3). From the hypothesis, it is evident that e is a shared variable of type T. If we create a reference to e, the type of $\&e$ is $*T^{\circ}$.

$$\text{Read}: \frac{S \vdash e: \ T \ M}{S \vdash e: \ T^{\circ}} \tag{2.4}$$

This is the typing rule for `Read` (2.4). It means that if we read from an expression e, we will obtain an exclusive value, which is straightforward because reading from the expression e retrieves the mathematical value stored in the location associated with e.

$$\text{Write}: \frac{S \vdash e: \ T \ M}{S \vdash e: \ T^{\circ}} \tag{2.5}$$

This is `Write` rule (2.5). It means that when we write to the expression e, we just put the mathematical value into the related memory location.

$$\text{GetField}: \frac{S \vdash e: \ \mathit{Struct} \ M \quad \mathit{Struct}.f : T}{S \vdash e.f: \ T \ M} \tag{2.6}$$

This is `GetField` typing rule (2.6). It indicates that the modifiers of the fields within a struct type are consistent with the modifier of the struct itself.

$$\text{Index}: \frac{S \vdash e: \ [n]T \ M \quad S \vdash e' : \mathit{int}^{\circ}}{S \vdash e.f: \ T \ M} \tag{2.7}$$

This is `Index` typing rule (2.7). It means that the modifiers of the elements within an array are consistent with the modifier of the array.

$$\text{FunctionCall}: \frac{S \vdash e_1: \ T_1^{\circ} \ ... \ S \vdash e_n: \ T_n^{\circ} \quad f : (T_1, ..., T_n) \mapsto T}{S \vdash f(e_1, ..., e_n) : T^{\circ}} \tag{2.8}$$

This is `FunctionCall` typing rule (2.8), and it shows that all the arguments and the return values of a function are exclusive values.

2.3.2 Statement Judgment

Below are the main statement judgments in Gobra. Since statements do not produce values, the typing rules take the form of $S \vdash s$, where s does not possess a type.

$$\text{Sequence}: \frac{S \vdash s_1 \quad S \vdash s_2}{S \vdash s_1; s_2} \tag{2.9}$$

This is `Sequence` rule (2.9). It shows that if s_1 and s_2 are valid in given the context S, the sequence of $s_1; s_2$ should also be valid.

$$\text{Declare}: \frac{S[x \mapsto T\ M] \vdash s}{S \vdash var\ x\ M\ T\ in\ s} \tag{2.10}$$

This is `Declare` rule (2.10). In the premise, it shows that given context S, s is valid. Thus, declaring a variable x in the form of $var\ x\ M\ T\ in\ s$ is also valid.

$$\text{Assignment}: \frac{S \vdash e:\ T^{\circ} \quad S \vdash e':\ T^{\circ}}{S \vdash e = e'} \tag{2.11}$$

This is `Assignment` rule (2.11). It means it is valid to assign the expression e' of type T to the expression e. It's worth mentioning that since loading from/to the locations is required during the assignment, both e and e' are exclusive. Additionally, the premise of the assignment essentially consists of two expression judgments.

$$\text{If}: \frac{S \vdash e:\ bool^{\circ} \quad S \vdash s_1 \quad S \vdash s_2}{S \vdash if\ e\ \{s_1\}\ else\ \{s_2\}} \tag{2.12}$$

This is `If` rule (2.12). It indicates that if e has the type of $bool^{\circ}$, and s_1 and s_2 are valid statements, $if\ e\ \{s_1\}\ else\ \{s_2\}$ is the valid form of `if-else` control flow.

$$\text{Loop}: \frac{S \vdash e:\ bool^{\circ} \quad S \vdash s}{S \vdash while\ e\ \{s\}} \tag{2.13}$$

This is `Loop` rule (2.13). It is similar to `If` rule (2.12) and it gives the legal format of a `loop` procedure.

Chapter 3

Linear Type Systems

In Chapter 2, we demonstrate how to verify heap-allocated values using permission reasoning in Gobra, ensuring the safety of data modification. Nevertheless, permission reasoning comes with substantial costs. As elucidated by Wolf et al. [1], heightened usage of memory locations tracked with permissions may increase Gobra's verification time due to the increased computational load on the backend solver. In extreme cases, verification might even fail due to the non-termination of the verification process. Permission reasoning imposes an annotation overhead and contributes to the elongation of verification time. In this chapter, we focus on the necessary background knowledge to understand linear type systems.

In a linear type system[1], the concepts of *no duplication* and *no discard* are fundamental principles that govern how resources are managed and utilized. These principles are aimed at enforcing strict control over how variables are used and ensuring efficient resource utilization [3].

The principle of *no duplication* ensures that, at any given time, a single reference is associated with each memory location. This eliminates the occurrence of aliasing, ensures that the ownership of resources remains clear, and prevents issues such as multiple pointers referring to the same memory block and causing unexpected modifications. Moreover, this principle empowers the compiler to execute more aggressive optimizations, thereby enhancing program performance. *No discard* dictates that a resource must be used exactly once and cannot be discarded prematurely or left unused, and this principle mandates that programmers bear the responsibility of intentionally releasing memory. In other words, when a resource is allocated, the program is obligated to consume it before moving forward. This prevents unnecessary use of computational resources, like memory, and guarantees

[1]In this section, we refer to a strict linear type system.

that all allocated resources are put to use effectively, and effectively sidesteps the need for garbage collection [3].

Figure 3.1 shows a sample code in a linear type system. We use `x!` to indicate that variable `x` is a linear type, and in line 3 we want to create a reference `p` pointing to `x`, which is not allowed in linear type systems because of the no duplication property. Line 5 is needed - the program needs to consume variable `x` before the end of the function `foo`.

```
1  func foo() {
2      var x! int = 42
3      var p *int = &x // not allowed in strict linear type systems
4                      // because of the "No duplication" property
5      x! = nil        // needed because of the "No discard" property
6  }
```

Figure 3.1: No duplication and no discard

In summary, no discard ensures that resources are not wasted and are fully utilized, while no duplication maintains the uniqueness of resources and prevents unintended sharing or copying. These principles play a crucial role in enhancing resource safety and program efficiency within a linear type system.

Some aforementioned studies [4] [6] indicate that linear type systems are a more efficient resource management approach. Unlike permission-based approaches where programmers need to explicitly define and manage permissions for each location, linear type systems enforce the correct usage and access control of memory locations through the type system itself. Thus, using linear type systems to replace permission reasoning to simplify memory reasoning is feasible. As research in linear type systems continues to expand, various variants of linear type systems have emerged. In the next section, we will introduce the borrowing mechanism used in the Rust programming language, which is one of these variants.

3.1 Borrowing in Rust

Modern linear type systems offer more advanced capabilities. For instance, it could be possible to create references to linear-typed values under certain constraints. Rust's *borrowing* mechanism exemplifies this idea. This section will delve into the concept of borrowing in Rust, which has significantly inspired the type system developed in this thesis [7].

In Rust, each location has an *owner*. A location can only be utilized within the scope of its owner. The ownership of a location is transferred between functions by passing them as arguments and return values. For instance, as shown in Figure 3.2, the string variable `s1` is declared within the `main` function, consequently endowing the function `main` with ownership of `s1`. When `main` invokes the `length` function, the ownership of variable `s1` is transferred to the function `length`. If we intend to use this string variable in `main` after this function call, we need to return the ownership of `s1` as an output value, which is assigned to `s2`.

```
1  fn main() {
2      let s1 = String::from("hello");
3      let (s2, length) = length(s1);
4  }

5  fn length(s1: String) -> (String, usize) {
6      let length = s1.len();
7      (s1, length)
8  }
```

Figure 3.2: Rust code without using borrowing

```
1  fn main() {
2      let s1 = String::from("hello");
3      let length = length(&s1);
4  }

5  fn length(s: &String) -> usize {
6      s.len()
7  }
```

Figure 3.3: Rust code using borrowing

Borrowing allows programmers to create references to the same memory location, thereby granting the reference the ability to read or modify the location within a specific scope without transferring ownership. When the borrowing scope is exceeded, which is determined by Rust's compiler automatically, the borrowing will become unusable. Unlike ownership, borrowing supports concurrent access by multiple parts of the program while maintaining strict rules to prevent data races and ensure data integrity. This mechanism strikes a balance between safety and flexibility, allowing for efficient resource utilization and manipulation without sacrificing the strong guarantees provided by Rust's ownership system. Figure 3.3 implements the same functionality as in Figure 3.2. In line 3, however, we create a borrowing of `s1` and pass this borrowing into the `length` function. Notably different from before, in line 5, the `length` function returns only the length of the given string, excluding the string itself. This is a consequence of borrowing, which ensures that the `length` function can only temporarily utilize `s1` without taking the ownership of it. The scope of this borrowing is within the body of the `length` function. When the function ends, this borrowing `&s1` becomes unavailable.

Section 2.2.2 presents that each reference possesses either read-only permission or full permission. Multiple references with read-only permissions can coexist, but references with full permission cannot coexist with any other reference. This idea closely aligns with Rust's borrowing mechanism, where there are *mutable borrowings* and *immutable borrowings*, reflecting a similar concept.

In Rust, both mutable and immutable borrowings are references that allow a code snippet to temporarily access and interact with the related locations. These borrowings ensure safe and controlled concurrent access to memory locations. *Immutable borrowings* allow multiple parts of the program to read data concurrently without the risk of modification, denoted as `&`. This is particularly useful when the programmers want to share data for reading purposes while preventing accidental changes. *Mutable borrowings*, on the other hand, enable exclusive access to a location for modification. It ensures that only one thread can modify the location at a time, identified as `&mut`. This prevents data races and concurrent modifications that could lead to unexpected behavior.

```
1  fn main() {
2      let vec = vec![1, 2, 3];
3      let iborrow = &vec;      // Immutable borrowing
4      println!("iborrow[0]", iborrow[0]); // Reading data

5      let mborrow = &mut vec; // Mutable borrowing
6      mborrow.push(4);         // Modifying data
7  }
```

Figure 3.4: Two kinds of borrowing

In Figure 3.4, line 3 creates an immutable borrowing `iborrow` of `vec`, which has only read access to `vec`. Line 5 creates a mutable borrowing `mborrow` of `vec`, which has write access to `vec` and can update vec in line 6.

It is worth mentioning that even though a linear type system is efficient in resource and memory management, it also has some limitations. For instance, it is hard and even impossible to express cyclical data structures in safe Rust due to the language's strict ownership and borrowing rules. Rust's borrowing system prohibits multiple mutable references to the same location, which is required for creating cyclic data structures.

Chapter 4

Design

This chapter demonstrates our design of the linear type system for Gobra. Section 4.1 presents the overview of the type system, where we introduce the key concepts of our type system. Section 4.2 introduces the newly added type modifiers and two kinds of new references in the system, inspired by Rust. Section 4.3 presents the typing of expressions in our linear type system. Section 4.4 details the well-definedness of statements in our type system. Section 4.5 discusses the new *borrowing context*, used in the type checking process. Section 4.6 demonstrates how shared variables interact with the new type system. Section 4.7, 4.8, and 4.9 show how calls, interfaces, and recursive structs are type-checked. Lastly, Section 4.10 summarizes the syntax of the new type system.

4.1 Overview

To introduce linearity to Gobra, we add two type modifiers to the system besides shared and exclusive in Gobra's type system, that is *linear write* and *linear read*, denoted as `!` and `?`, respectively. We refer to types with linear read or linear write modifiers as *linear* types and similarly refer to values and locations with a linear type as *linear* values and *linear* locations, respectively. *Linear* values are similar to shared values because the linear values are also addressable, yet they also possess distinct characteristics. As mentioned in Section 2.2.2, all accesses to shared variables must be proven safe through permission-based reasoning. Conversely, in the case of a linear variable, the type system takes the burden of justifying that a location is accessible. We refer to the type system augmented with the linear modifiers as the *linear type system* for Gobra.

Similar to Rust, creating a reference imparts access to the associated location. Therefore references pointing to linear locations possess either read-only access or write access. Whenever we access or create a reference, our type system must guarantee the safety of the operation.

As previously discussed, the linear type system is responsible for managing the accesses of linear locations and ensuring memory safety. Hence, it becomes necessary to record information about all references to linear variables. In the linear type system, we introduce a novel context known as *the borrowing context*. This context captures and maintains all references associated with linear variables, and is used by the linear system to determine whether an operation is allowed or not.

Figure 4.1 is a code snippet in Gobra without linearity and permission reasoning is needed for memory locations. In the function `foo`, we declare a shared variable `x` that resides on the heap. In line 3, we create a reference `y` that points to `x` and pass `y` as an argument to the function `add5`. To safely update `y` within the function `add5`, we transfer the write permission of `y` to the function `add5` using the precondition, denoted as `requires acc(y)`. Then the permission is returned back to the caller through the postcondition, denoted as `ensures acc(y)`.

Figure 4.2 shows how we can eliminate permission reasoning through the application of the linear type system. In line 2, we declare a linear write variable `x`. Line 3 creates a reference `y` to `x` and indicates that `y` has write access to `x` with `&w`. The syntax used here will be detailed in Section 4.2. In line 4, we pass `y` to the function `add5` without the need for permission reasoning. Since `y` has write access to the associated memory, the linear type system determines that the function `add5` can update `y` safely.

```
1   func foo() {
2       var x@ int = 42
3       var y *int = &x
4       add5(y)
5   }
6   requires acc(y)
7   ensures acc(y)
8   func add5(y *int) {
9       *y += 5
10  }
```

Figure 4.1: Example code of using permission reasoning

```
1   func foo() {
2       var x! int = 42
3       var y *int = &w x
4       add5(y)
5   }
6   requires y != nil
7   func add5(y *w int) {
8       *y += 5
9       // safely update
10  }
```

Figure 4.2: Example code of using a linear type system

4.2 Type Modifiers and References

We introduce two type modifiers, linear write and linear read, to Gobra. As their names imply, linear write locations have the ability to modify the related memory, whereas linear read locations can only read values stored in the associated memory. Since the type system has several possible modifiers with different accesses to a memory location, we need to know which kind of access the programmers want when referencing a location. Therefore, in our linear type system, references to linear locations are also categorized into two types: *read references* and *write references*, represented as `*r T` and `*w T` separately (`T` is the type of the value that the reference points to). Read references are used to read from the location they point to. However, they cannot be used to write to these locations. Write references are used both for reading and writing. Similar to immutable borrowings and mutable borrowings in Rust, in our linear type system, for a linear location, we can also have multiple read references pointing to it at the same time or we can have one write reference. If a write reference exists to a memory location, no read references can co-exist at that point in time.

It is also worth noting that we can create both read and write references to linear write locations, whereas only read references can be created for linear read locations. The access held by a reference cannot exceed the access owned by the linear location itself. We use `&r` to create read references and use `&w` to create write references.

Figure 4.3 shows an example demonstrating how to create these two kinds of references to linear variables. Line 1 declares a linear write variable `x`. Line 2 creates a read reference to `x`, and line 3 creates a write reference to `x`, which are allowed in our type system. In line 4, we declare a linear read variable `y`. Line 5 creates a read reference to `y`, which is also valid. However, if we want to create a write reference to `y` or modify `y`, the type system will report errors because the accesses required for these two operations exceed what `y` possesses.

```
1  var x! int = 10        // linear write variable
2  var rp *r int = &r x   // read reference
3  var wp *w int = &w x   // write reference

4  var y? int = 40        // linear read variable
5  var rq *r int = &r y   // read reference
6  var wq *w int = &w y   // not allowed!!!
7  y = 50                 // not allowed!!!
```

Figure 4.3: Example code of new features

4.3 Expression Judgment

We have introduced two type modifiers into Gobra, and also split the references to linear locations as read references and write references. To track the linearity and reference information, we introduce the borrowing context into the linear type system. Therefore, it is necessary to modify the typing judgments accordingly. Table 4.1 introduces the notation used in the judgments shown in the upcoming sections.

Symbol	Meaning
$\circ$	Exclusive Modifier
@	Shared Modifier
?	Linear Read Modifier
!	Linear Write Modifier
M	Modifier, $M := \circ \mid @ \mid ! \mid ?$
Lin	$Lin := ! \mid ?$
$NoLin$	$NoLin := \circ \mid @$
T	Type
S	Type Context, $S: \ \mathit{variable} \mapsto T\ M$
X	Borrowing Context
C	Combined Context
e	Expression
x	Variable
t	target of reading expression judgment, can be _
s	statement

Table 4.1: Notation used in typing judgements - Linear Gobra

Same as before, $\circ$ and @ are exclusive and shared modifiers. The annotations ? and ! denote the linear read and linear write modifiers, respectively. The symbol M is the modifier of a location, which could potentially be any of the four modifiers. The symbol Lin denotes a linear modifier, while the symbol $NoLin$ denotes that a location is not linear. S represents the type context, which maps variables to their types (T) and modifiers (M). We use X to represent the borrowing context. Lastly, C represents the combined context of S and X.

Expression judgments are split into *reading expression judgments* and *writing expression judgments*. We respectively abbreviate them as reading judgments and writing judgments. These two judgments are necessary since the type systems demand different accesses, namely read and write accesses, depending on whether the typed expression is the target of an assignment or not.

Writing judgments are utilized for assignments' left-hand side while reading judgments are applicable to all other scenarios. The read judgment additionally adds a new element, target t, which is the target of the assignment in case the typed expression is on the right-hand side of an assignment. If the typed expression is not read on the right-hand side of an assignment, there is no target, denoted as $\_$. Writing judgments do not have targets. Syntactically, we distinguish between writing and reading judgments based on whether the judgment has a target or not.

4.3.1 Pre-Context and Post-Context

Because the borrowing context records the information of references to all linear locations, certain operations like creating and dereferencing a reference can potentially modify the borrowing context. To address this, we introduce the concepts of *pre-contexts* and *post-contexts*. As the names imply, the pre-context refers to the borrowing context before evaluating an expression, while the post-context is the context after an expression has been evaluated.

For example, below is the reading judgment of `ReadLinearDereference`. We note that since the expression e potentially involves creating a read or write reference to a linear location, the pre-context X becomes the post-context X' after evaluation.

$$\frac{X; S; t \vdash e : \; *(T \; Lin)^{\circ} \mid X' \quad \mathit{read\_activate}(X', \; e) = X''}{X; S; t \vdash *e : \; T \; Lin \mid X''}$$

4.3.2 Read Expression Judgment

In this section, we outline the main reading judgments, as shown below. We will provide explanations for the crucial typing rules.

$$\text{ReadLinearDereference} : \frac{X;S;t \vdash e : \; *(T \; Lin)^{\circ} \mid X' \quad \mathit{read\_activate}(X', \; e) = X''}{X;S;t \vdash *e : \; T \; Lin \mid X''} \tag{4.1}$$

The `ReadLinearDereference` typing rule (4.1) was shown in the previous section. It is known from the premise that e is a read or write reference, denoted by $*(T \; Lin)^{\circ}$. The type system needs to ensure that the dereference operation is safe, which is guaranteed by a side condition that the expression e can be *activated* in the borrowing context X'. *Activating* a reference means to obtain the related access possessed by the reference. `read_activate` is the operation to activate the read access of a given reference. More detailed explanations about operations on the borrowing context will be provided in section 4.5. Thus, the borrowing context changes to X'' after we dereference e.

$$\text{ReadShared}: \frac{X;S;t \vdash e:\ T\ @ \mid X'}{X;S;t \vdash e:\ T\ \circ \mid X'} \tag{4.2}$$

This is the `ReadShared` rule (4.2). It indicates reading from a memory location that stores a shared value of type T results in an exclusive result of type T since reading from this location yields a mathematical value.

$$\text{ReadLinear}: \frac{X;S;t \vdash e:\ T\ Lin \mid X' \quad read\_activate(X',\ \&e) = X''}{X;S;t \vdash e:\ T\ \circ \mid X''} \tag{4.3}$$

This is the rule for `ReadLinear` (4.3). Reading from a memory that stores a linear value of type T (either linear read or linear write) will yield an exclusive result of type T. Similar to the `LinearDereference` rule (4.1), the type system needs to justify that the read is safe with the side condition $read\_activate(X',\ \&e) = X''$.

$$\text{SharedReference}: \frac{X;S;t \vdash e:\ T\ @ \mid X'}{X;S;t \vdash \&e:\ *(T@)\ \circ \mid X'} \tag{4.4}$$

This is the `SharedReference` typing rule (4.4), which creates a reference to a shared location. This rule indicates that creating a reference to a shared location will get a reference of type $*(T@)\ \circ$.

$$\text{LinearReadReference}: \frac{X;S;t \vdash e:\ T\ \ Lin\ \mid\ X' \quad push\_read\_ref(X',\ \&r\ \ e,\ t) = X''}{X;S;t \vdash \&r\ \ e:\ *(T?)\ \circ\ \mid\ X''} \tag{4.5}$$

This is the `LinearReadReference` rule (4.5), which creates a read reference to a linear location. We use $*(T?)\ \circ$ to represent a read reference to a linear location of type T. The linear type system justifies that the operation($\&r\ e$) is safe with the condition $push\_read\_ref(X',\ \&r\ \ e,\ t) = X''$. The operation `push_read_ref` creates a read reference to e within the borrowing context. Thus, the borrowing context changes from X' to X'' due to the operation($\&r\ e$).

$$\text{LinearWriteReference}: \frac{X;S;t \vdash e:\ T\ \ !\ \mid\ X' \quad push\_write\_ref(X',\ \&e,\ t) = X''}{X;S;t \vdash \&w\ \ e:\ *(T!)\ \circ\ \mid\ X''} \tag{4.6}$$

This is the `LinearWriteReference` typing rule (4.6). It is similar to the `LinearRead- Reference` typing rule (4.5). The distinction lies in the fact that we can only create write references to linear write variables. We use $*(T!)\ \circ$ to represent a write reference to a linear write location of type T. Similarly, we use the operation `push_write_ref` to create a write reference to e within the borrowing context, which changes the borrowing context from X' to X''.

The following three judgments are similar to Gobra's old expression judgments. It is important to note that every expression e has the potential to modify the borrowing context.

$$\text{ReadVariable}:\frac{}{X;S;t\vdash x:\ T\ M\mid X} \tag{4.7}$$

$$\text{ReadGetField}:\frac{X;S;t\vdash e:\ \mathit{Struct}\ M\mid X' \quad \mathit{Struct}.f:T}{X;S;t\vdash e.f:\ T\ M\mid X'} \tag{4.8}$$

$$\text{ReadIndex}:\frac{X;S;t\vdash e:\ [n]T\ M\mid X' \quad X';S;_{\vdash} e':\ \mathit{int}\ {}^\circ\mid X''}{X;S;t\vdash e[e']:\ T\ M\mid X''} \tag{4.9}$$

4.3.3 Writing Expression Judgment

In this section, we outline the main writing judgments and provide explanations for the crucial ones.

$$\text{WriteLinearDereference}:\frac{X;S;\_\vdash e:\ *(T\ !)^\circ\mid X' \quad \mathit{write\_activate}(X',\ e){=}X''}{X;S\vdash *e:\ T\ !\mid X''} \tag{4.10}$$

This rule is similar to the `ReadLinearDereference` rule (4.1) in the reading judgments, but it dereferences a write reference e. The target in the premise changes to _. This is because, in the hypothesis, we still need to read from the reference e to get the address it points to. The expression e is read without being assigned to a target, so the target changes to _. In the conclusion, we dereference e and the result $*e$ is on the left-hand side of an assignment. Thus, the target in the conclusion is eliminated. Additionally, we use the *write_activate* operation to activate the write access possessed by e, changing the borrowing context from X' to X''.

$$\text{WriteShared}:\frac{X;S\vdash e:\ T\ @\mid X'}{X;S\vdash e:\ T\ {}^\circ\mid X'} \tag{4.11}$$

This is the rule for `WriteShared` (4.11), which is similar to `ReadShared` rule (4.2) in the previous section and requires no further elaboration at this point.

$$\text{WriteLinear}:\frac{X;S\vdash e:\ T\ !\mid X' \quad \mathit{write\_activate}(X',\ \&e)=X''}{X;S\vdash e:\ T\ {}^\circ\mid X''} \tag{4.12}$$

This is the `WriteLinear` rule (4.12). It means that we can only write to linear write locations and we use the operation `write_activate` to ensure that writing to the linear write location e is safe.

The three following judgments are similar to those in the previous section, except that the writing judgment does not have a target.

$$\text{WriteVariable}: \frac{}{X; S \vdash x:\ T\ M \mid X} \tag{4.13}$$

$$\text{WriteGetField}: \frac{X; S \vdash e:\ Struct\ M \mid X' \quad Struct.f : T}{X; S \vdash e.f:\ T\ M \mid X'} \tag{4.14}$$

$$\text{WriteIndex}: \frac{X; S \vdash e:\ [n]T\ M \mid X' \quad X'; S; \_ \vdash e':\ int\ ^{\circ} \mid X''}{X; S \vdash e[e']:\ T\ M \mid X''} \tag{4.15}$$

4.4 Statement Judgment

The meanings of most of the symbols used in this section can be found in Table 4.1, and we use s to represent a statement. Below are the new statement judgments in the type system.

$$\text{Sequence}: \frac{X; S \vdash s_1 \mid X' \quad X'; S \vdash s_2 \mid X''}{X; S \vdash s_1; s_2 \mid X''} \tag{4.16}$$

This is the `Sequence` rule (4.16), which is similar to the old `Sequence` rule (2.9). The only difference is that s_1 and s_2 might modify the borrowing context.

$$\text{NonLinearDeclare}: \frac{X; S[x \mapsto T\ NoLin] \vdash s \mid X'}{X; S \vdash var\ x\ NoLin\ T\ in\ s \mid X'} \tag{4.17}$$

This is the rule for `NonLinearDeclare` (4.17), which creates a non-linear variable x. This rule is quite similar to the `Declare` rule (2.10). The creation of a non-linear variable will not change the borrowing context, whereas the statement s has the potential to change the borrowing context, resulting in a change from X to X'.

$$\text{LinearReadDeclare}: \frac{create(X,\ \&x) = X' \quad X'; S[x \mapsto T?] \vdash s \mid X''}{X; S \vdash var\ x?\ T\ in\ s \mid X''} \tag{4.18}$$

This is the `LinearReadDeclare` rule (4.18). This rule declares a linear read variable x of type T, which means x has only read access to the memory location and cannot be modified afterwards. We use `create` operation to add linear information about x to the borrowing context, which changes from X to X'. Afterwards, the statement s assigns a value to x, changing the borrowing context to X''.

$$\text{LinearWriteDeclare} : \frac{create(X,\ \&x) = X' \quad X'; S[x \mapsto T!] \vdash s \mid X''}{X; S \vdash var\ x!\ T\ in\ s \mid X''} \tag{4.19}$$

This is the `LinearWriteDeclare` rule (4.19). This rule declares a linear write variable x of type T, which indicates x has write access to the memory location. We then use the statement s to assign a value to x. Similar to `LinearReadDeclare` rule (4.18), the borrowing context is X'' finally.

$$\text{Assignment} : \frac{X; S \vdash e :\ T^{\circ} \mid X' \quad X'; S; e \vdash e' :\ T^{\circ} \mid X''}{X; S \vdash e = e' \mid X''} \tag{4.20}$$

This is the rule for `Assignment` (4.20) in the linear type system. In the premise of the rule, we have a writing expression judgment for the expression e (without target) and a reading expression judgment for the expression e'. We will assign the value of e' to e, thereby the target of e' is e.

$$\text{If} : \frac{X;S;\_\vdash e\text{: } bool^{\circ} \mid X' \quad X';S\vdash s_1 \mid X'' \quad X';S\vdash s_2 \mid X''' \quad merge\_contexts(X'',\ X''')=X^m}{X;S\vdash if\ \ e\ \{s_1\}\ else\ \{s_2\} \mid X^m} \tag{4.21}$$

This is the judgment rule for the `If` (4.21) control flow. In the premise of this rule, we have a reading expression judgment for the expression e, which serves as the condition of the control flow, and two statement judgments for `if` and `else` branches, respectively. If the program enters the `if` branch, the borrowing context changes to X''. Otherwise, the borrowing context changes to X'''. We need to merge the borrowing contexts of these two branches, and the merged context is X^m. The operation used for merging different borrowing contexts is called `merge_context`, and it will be discussed in detail in Section 4.5.

$$\text{Loop} : \frac{W; S; \_ \vdash e :\ bool^{\circ} \mid W \quad W; S \vdash s \mid W \quad X \to W}{X; S \vdash while\ e\ \{s\} \mid W} \tag{4.22}$$

This is the judgment rule (4.22) for `Loop` and it is different from the previous judgments. We use W to represent the borrowing context and W denotes the *weakest* context that can keep pre-contexts and post-contexts unchanged after justifying $W; S; \_ \vdash e :\ bool^{\circ} \mid W$ and $W; S \vdash s \mid W$. Additionally, X in the conclusion implies W, meaning that W is *weaker* than X. The meaning of *weak* is that all reference relationships and linear information present in W will definitely appear exactly in X, but X may contain more information beyond W. The meaning of this judgment is that if there exists a W such that the premise holds, then the loop procedure *while* e $\{s\}$ with the pre-context X is valid.

4.5 The Borrowing Context

We introduce *the borrowing context* to the linear type system to track the linearity information, denoted as X. In Section 4.5.1, we will detail the data structure of the borrowing context. Section 4.5.2 introduces the main operations on the context. Section 4.5.3 will demonstrate how the judgments impact the borrowing context through a small example.

4.5.1 Tracking References with Stacks

As shown in Figure 4.4, We store information about all references using a map that maps memory locations to stacks. Each stack corresponds to a linear location known as *the borrowing stack*. The domain of these linear locations represents disjoint locations. The locations in a borrowing stack represent potential pointers to the linear location and cannot be `nil`. In other words, all write and read references in the borrowing context cannot be `nil`. A pointer `p` is in the borrowing stack of a linear location `l` if access to `l` is necessary to justify accessing `p`.

Each borrowing stack consists of a sequence of *frames*. A *frame* is the fundamental building block of the borrowing stack, where references are placed. We distinguish between two kinds of accesses, namely read and write accesses. Thus, the frames can also be categorized into two types based on access type: *read frames* and *write frames*. A read frame is a collection of locations with read access, while a write frame has only one location with write access. Only the top frame is active, the other frames are inactive. Being inactive does not necessarily prohibit access to the references within the frame. An inactive frame needs to be activated before accessing the references within it.

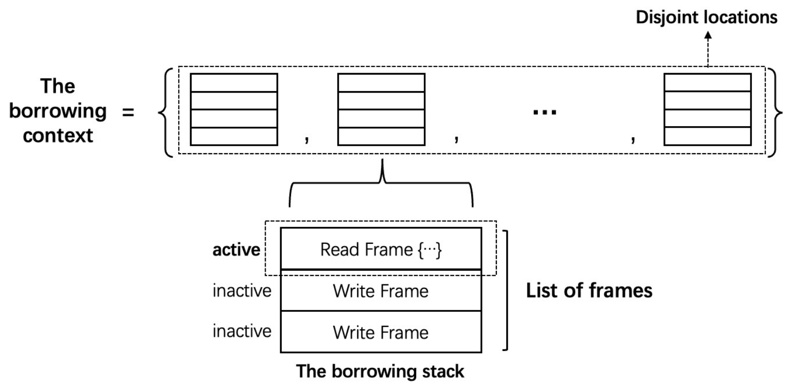


Figure 4.4: Schematic diagram of the borrowing context

4.5.2 Operations on the Borrowing Context

As discussed in Section 4.3 and Section 4.4, we have the following operations on the borrowing context:

`create`: Create a borrowing stack for a linear location and add a map from the linear location to the borrowing stack. If the location is linear write, the new borrowing stack will contain a write frame of the given location. Conversely, if the location is linear read, the new borrowing stack will contain a read frame that encompasses the given location. For example, if we declare a linear write variable `x`, we will create a borrowing stack for x. The changes in the borrowing context are shown in Figure 4.5.

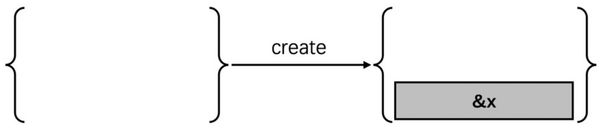


Figure 4.5: The schematic diagram of `create` operation

`push_write_ref`: Push a write reference pointing to a linear location into the borrowing context. Return `None` if this operation is not possible. This operation pushes a write frame containing the write reference to the related stacks. For example, if we create a write reference `p` to `x`, the changes in the borrowing context due to the operation `push_write_ref` are shown in Figure 4.6.

`push_read_ref`: Push a read reference pointing to a linear location into the borrowing context. Return `None` if this operation is not possible. For a certain borrowing stack, if the current top frame is a read frame, we just add the given new read reference as an element to the read frame. For example,

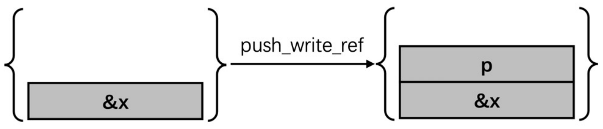


Figure 4.6: The schematic diagram of `push_write_ref` operation

the upper half of Figure 4.7 displays how the borrowing context changes if we push a read reference `t` into it. If the current top frame is a write frame, the situation becomes more complex. For example, when the current frame on top is a write frame having a write reference `p`, we need to push a read frame containing the given read reference `q` onto the borrowing stack. Additionally, we also need to add `p` into the read frame, as shown in the bottom half of Figure 4.7. The reason for doing so is intuitive. As mentioned earlier, a write reference possesses both write access and read access to a memory location. By performing this operation, we essentially separate the read access from the write reference. This ensures that if there is a need to read from the write reference `p`, we only need to activate the read access in the top read frame.

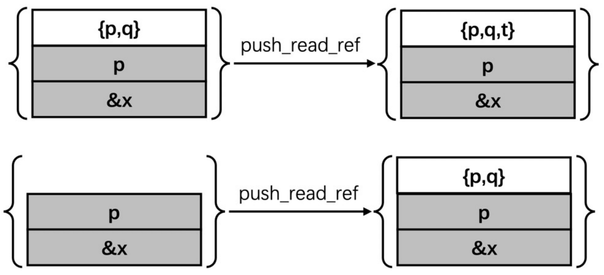


Figure 4.7: The schematic diagram of `push_read_ref` operation

`read_activate:` Pop frames to activate the read frames containing the given read reference in the borrowing context such that the read access of the given read reference is active. Return `None` if this operation is not possible. The changes in the borrowing context due to the `read_activate` operation to activate the read reference `q` are shown in Figure 4.8.

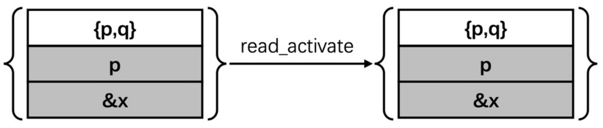


Figure 4.8: The schematic diagram of `read_activate` operation

`write_activate:` Pop frames to activate the write frames containing the given write reference in the borrowing context such that the write access of the given write reference to a linear location is active. Return `None` if this operation is not possible. For example, if we want to activate the write reference `p`, the changes in the borrowing context due to the `write_activate` operation are shown in Figure 4.9.

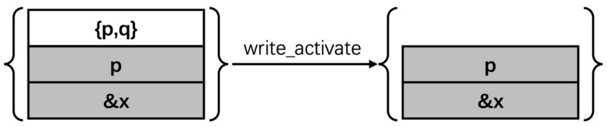


Figure 4.9: The schematic diagram of `write_activate` operation

`merge_context`: Merge different borrowing contexts. Utilizing different merge strategies can produce varying outcomes.

The code snippet below requires merging borrowing contexts from different branches due to the `if-else` control flow from line 5 to line 11. If the `if` branch is taken, the borrowing context should be as shown in Figure 4.10. Otherwise, Figure 4.11 shows what the borrowing context should look like if the `else` branch is taken.

```
1   var x! int = 5
2   var y! int = 6
3   var p *w int
4   var q *r int

5   if some_condition_known_at_runtime {
6       p = &w x
7       q = &r (*p);
8   } else {
9       p = &w y
10      q = &r (*p);
11  }
12  print!("{}",q);
13  *p=4;
14  print!("{}",p);
15  print!("{}",q);
```

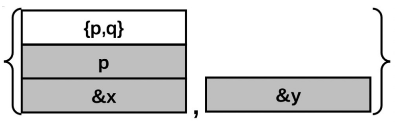


Figure 4.10: The borrowing context after executing `if` branch

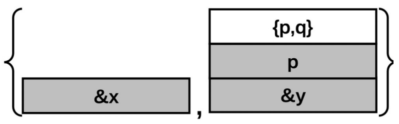


Figure 4.11: The borrowing context after executing `else` branch

We have two available merge strategies to merge the contexts, and we will demonstrate the difference between the two options. Option 1 entails that, for the corresponding borrowing stacks in the two contexts, if the *common tail* of the borrowing stacks matches one of the stacks, the merged result will adopt the larger stack. Otherwise, it will adopt the *common tail*. Option 2 adopts the *common tail* of all borrowing stacks present in the borrowing context. The *common tail* refers to the common frames from the bottom of two borrowing stacks. For instance, from Figure 4.10 and Figure 4.11, it is evident that the common tail of the borrowing stacks of `&x` is the bottom write frame containing `&x`. Similarly, the common tail of the borrowing stacks of `&y` is also the bottom write frame containing `&y`. For option 1, the common tail of `&x` matches the borrowing stack of `&x` in Figure 4.11, so the merged result of `&x` is the borrowing stack in Figure 4.10 (the larger stack). The rest merge operations are quite similar, and we will not elaborate further on them here.

The merging outcomes of the two options are illustrated in Figure 4.12. If we opt for option 1, the references `p` and `q` are present in multiple borrowing stacks. Removing reference `p` from one borrowing stack necessitates its removal from all the stacks in which reference `p` is present. Activating reference `p` also requires activating `p` in all borrowing stacks. For example, line 13 needs to `write_activate` reference `p`, so the top read frames of stack `&x` and `&y` will both be removed.

However, if we select option 2, references `p` and `q` will no longer be accessible after line 11. In this case, the linear type system will report an error in line 12 since reference `q` is inaccessible.

Thus, we need to make a trade-off between the complexity of the linear type system and its expressiveness. Consequently, we opt for option 1.

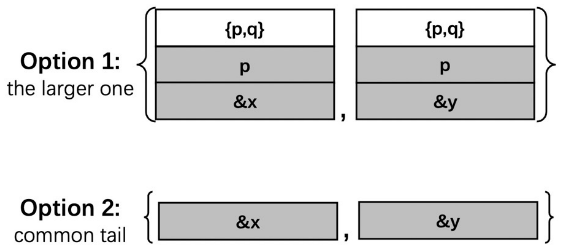


Figure 4.12: Two options for merging contexts

Finally, it is worth noting that in the subsequent code demonstrations, we will use $\&a \mapsto [\{Read\_Ref\_Set\} \ \# \ Write\_Ref \ \# \ ...]$ to represent a borrowing stack of location $\&a$ and use $\{\&a \mapsto [\ ],\ \&b \mapsto [\ ], ...\}$ to denote the borrowing context.

4.5.3 Illustrating the Application of Judgments

As shown in Figure 4.13, `x` is a linear write variable, `wp` is a write reference to `x`, `m` is an exclusive int variable. Thus, the S context should be $S = \{x \mapsto int!, m \mapsto int^{\circ}, n \mapsto int^{\circ}, wp \mapsto *(int!)^{\circ}, rp \mapsto *(int?)^{\circ}\}$.

```
1  var x! int in x = 5;
2  var wp (*w int) in wp = &w x;
3  var m int in m = *wp;
4  *wp = m;
```

Figure 4.13: Example code for type checking

Line 1 declares a linear write variable `x`. Rules `LinearWriteDeclare` (4.19) and `Assignment` (4.20) are applied for typing. The `WriteLinear` rule is applied for expression judgment, at meanwhile the borrowing context changes from {} to $\{\&x \mapsto [\&x]\}$, as shown below.

$$\begin{array}{r|l}
x = 5; & \text{WriteLinear} : \dfrac{X;S \vdash x{:}int! \mid X'}{X;S \vdash x{:}int^{\circ} \mid X'} \quad X == X' \\
 & \text{Assignment} : \dfrac{X;S \vdash x{:}int^{\circ} \mid X' \quad X';S;x \vdash 5{:}int^{\circ} \mid X''}{X;S \vdash x{=}5 \mid X''} \\
 & X == X' == X'' \\
var\ x!\ int\ in\ x = 5; & \text{LinearWriteDeclare} : \dfrac{create(X,\ \&x){=}X' \quad X';S \vdash x{=}5 \mid X''}{X;S \vdash var\ x!\ T\ in\ x{=}5 \mid X''} \\
 & X == \{\}, X' == \{\&x \mapsto [\&x]\}, X'' == \{\&x \mapsto [\&x]\}
\end{array}$$

Then in line 2, we create a write reference to `x`. For this line, we use `NonLinearDeclare` (4.17) and `Assignment` (4.20) rules for statement typing, and apply `LinearWriteReference` rule (4.6) when creating `wp`. For the sake of brevity, we do not show the use of `NonLinearDeclare` and `Assignment` rules. The use of expression judgment `LinearWriteReference` is shown below, and the borrowing context changes from $\{\&x \mapsto [\&x]\}$ to $\{\&x \mapsto [wp \ \# \ \&x]\}$.

$$\begin{array}{r|l}
\&w\ x & \text{LinearWriteReference} : \\
 & \dfrac{X;S;wp \vdash x{:}int! \mid X' \quad write\_reference(X',\ \&w\ x,\ wp){=}X''}{X;S;t \vdash \&w\ x{:}\ *(int!)^{\circ} \mid X''}
\end{array}$$

We utilize `*wp` for both reading and writing in lines 3 and 4. Rules `ReadLinearDereference` (4.1) and `WriteLinearDereference` (4.10) will be applied, respectively. The use of typing rules is similar to the previous ones and will not be shown here.

4.6 Conversion

In certain scenarios, we might want to convert a linear write location to a shared location. In such cases, we can use the `&toS` operation for this kind of conversion. It is worth noting that after the conversion, we need to remove the borrowing stack of the linear write variable from the borrowing context using the `remove` operation. The result of this conversion is an exclusive value of type T, which is used for assigning to a shared variable (the target t, which is required to be shared).

$$\text{ConvertToShared}: \frac{X;S;t \vdash e:\ T! \mid X' \quad remove(X',\ \&e) = X''}{X;S;t \vdash \&toS\ e:\ T^{\circ} \mid X''} \tag{4.23}$$

Similarly, if we want to convert a full-permission shared location to a linear write location, we can use `&toL` to perform this conversion. The full permission of the shared location will be transferred to the converted result (the target t, which is required to be linear write). Additionally, the borrowing stack of the converted result will be added to the borrowing context with the help of the `create` operation.

$$\text{ConvertToLinear}: \frac{X;S;t \vdash e:\ T@ \mid X' \quad create(X',\ \ t) = X''}{X;S;t \vdash \&toL\ e:\ T^{\circ} \mid X''} \tag{4.24}$$

Below is an example showing how to perform conversions between linear write and shared values. The function `ConvertToShared` converts a linear write variable `x` to a shared variable `y`. The borrowing context changes from $\{\&x \mapsto [\&x]\}$ to $\{\}$ accordingly. And the assertion `assert acc(&y)` holds in line 6. The function `ConvertToLinear` receives a reference `x` and obtains the full permission of `x` by the precondition `requires acc(x)`. In line 10, `x` is converted to a linear write variable `y`. The borrowing stack of `y` will be added to the borrowing context. Thus, the borrowing context is $\{\&y \mapsto [\&y]\}$. The permission owned by `x` cannot be transferred back through postcondition.

```
1   func ConvertToShared(){
2       var x! int = 10
3       // {&x -> [&x]}
4       var y@ int = &toS(x)
5       // {}
6       assert acc(&y)
7   }

8   requires acc(x)
9   func ConvertToLinear(x *int){
10      var y! int = &toL(x)
11      // {&y -> [&y]}
12  }
```

4.7 Function Call

Due to the introduction of the borrowing context and the support for the conversion mechanism between shared values and linear values, the typing rule for function calls becomes considerably intricate.

Figure 4.14 is an example of why the previous `FunctionCall` (2.8) judgment is difficult to extend to the linear type system. Line 1 declares 3 linear write variables x, y, and z. Line 2 illustrates the borrowing context at that point: $\{\&x \mapsto [\&x], \&y \mapsto [\&y], \&z \mapsto [\&z]\}$. Then in line 3, we call the function f and pass the write references of x and y into f. We separate the checking of the linear type system from verification, so it is challenging to determine the borrowing context after the function call only based on the signature of the function f, which is shown in line 4.

If c is a reference pointing to variables declared inside the function f, we need to add the borrowing stack of p to the borrowing context after the call since the return value c is assigned to p. However, if c is the reborrowing of references a or b, we cannot add the borrowing stack of p to the borrowing context. This is because the locations in the borrowing context should be disjoint. In a more challenging scenario, the accesses passed in the function may not be transferred back due to internal uncertainty. For example, if we convert a and b to shared values, x and y should become inaccessible after the function call and we need to remove the borrowing stacks of x and y from the borrowing context. The analysis shows that the borrowing context is uncertain after a function call.

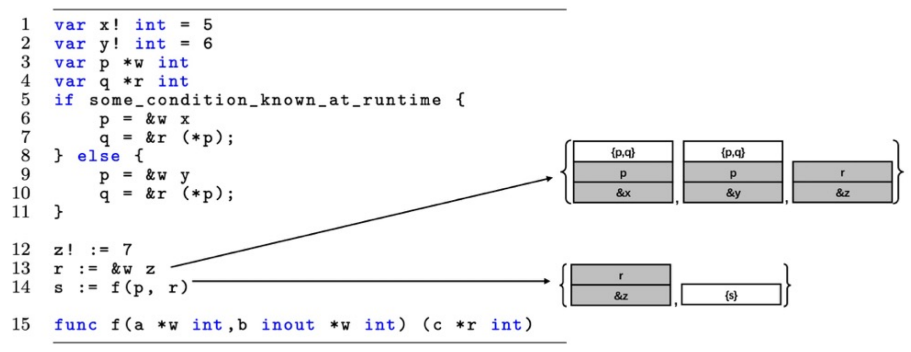
```
1  x! := 5; y! := 6; z! := 7
2  // X: {&x -> [&x], &y -> [&y], &z -> [&z]}

3  p := f(&w x, &w y)
4  // the signature of function f: f(a, b *w int) (c *r int)
```

Figure 4.14: Example code of function call

Thus, we introduce a new keyword *inout* to address this problem. The *inout* keyword annotates read or write references of a function's parameters or receivers, and it represents that the read or write references passed into the function should remain active after the function's execution. The *inout* keyword ensures that the accesses possessed by read or write references will definitely be transferred back, which helps to determine the borrowing context after function calls.

Figure 4.15 shows an example of using inout keyword. Lines 1 to 11 are the same codes illustrated in Section 4.5.2. Then we create a linear write variable z and a write reference r. Thus, the borrowing context at line 13 is $\{\&x \mapsto [\{p, q\} \ \# \ p \ \# \ \&x], \&y \mapsto [\{p, q\} \ \# \ p \ \# \ \&y], \&z \mapsto [r \ \# \ \&z]\}$, as shown in Figure 4.15.

Then in line 14, we invoke function f(p, r). The signature of the function f is shown in line 15. Since the parameter a is not annotated with inout, the access possessed by reference p will not be transferred back. Thus, the borrowing stacks containing p, the borrowing stack of x and y, are removed from the borrowing context. Since b is annotated with inout, the borrowing stacks containing the write reference r stay unchanged. Because



```
var x! int = 5
var y! int = 6
var p *w int
var q *r int
if some_condition_known_at_runtime {
    p = &w x
    q = &r (*p);
} else {
    p = &w y
    q = &r (*p);
}

z! := 7
r := &w z
s := f(p, r)

func f(a *w int,b inout *w int) (c *r int)
```

Figure 4.15: Example code of using inout

the return value of function `f` is a read reference and is assigned to `s`, we add a borrowing stack of `s`. Finally, the borrowing context changes to $\{\&z \mapsto [r \# \&z], s \mapsto [\{s\}]\}$, as shown in Figure 4.15.

4.8 Interface

As mentioned in the previous section, the inout keyword can also be used to annotate receivers of methods. Figure 4.16 shows how to annotate receivers with inout keyword. In line 8, we can see the receiver of the method `inc` is a write reference to a linear location, and the receiver is annotated with inout keyword. Line 9 shows that the receiver of the method `value` is a read reference, which is also annotated with inout. The inout keyword can ensure that after invoking the method `inc`, the caller is still active.

For interfaces, there are not many changes required. We also track the linearity information of interfaces. The casts between subtypes and interfaces are always treated like function calls, and the subtypes and interfaces are required to have the same accesses.

For example, in Figure 4.16, we declare an interface `Counter` from line 1 to line 4. Lines 5 to 7 declare a subtype of `Counter`, namely `Cell`. From line 8 to line 13, the type `Cell` implements the two methods of the interface `Counter`. The method `inc` is used to increase `cell.f` by 1, and the method `value` returns `cell.f` directly.

Then we have a `client` function. We declare a read reference `x` pointing to a linear `Cell` in line 15. The borrowing context is $\{x \mapsto [x]\}$. Then `x` is cast to `y`, the interface `Counter`, in line 17. We treat this cast as a function call, which receives a subtype instance (without inout keyword) and returns an interface instance (e.g., the `toInterface(x)` function). Thus,

```
 1  type Counter interface {
 2      inc()
 3      value() int
 4  }

 5  type Cell struct {
 6      f int
 7  }

 8  func (cell inout *w Cell) inc() {
 9      cell.f += 1
10  }

11  func (cell inout *r Cell) value() int {
12      return cell.f
13  }

14  func client() {
15      x := &w Cell{0}
16      // {x -> [x]}
17      var y Counter<w> = toInterface(x)
18      // {y -> [y]}
19      y.inc()
20      var z *w Cell = y.(*w Cell)
21      // {z -> [z]}
22      z.inc()
23      // {z -> [z]}
24      println(z.value()) // 2

25      y.inc() // fails
26  }
```

Figure 4.16: Example code of interface

we remove the borrowing stack of `x` from the borrowing context and add the borrowing stack of `y` into the borrowing context. Because the access of `y` is the same with `x` (indicated by `Counter<w>`), the borrowing stack of `y` is $y \mapsto [y]$. Finally, the borrowing context changes to $\{y \mapsto [y]\}$, as shown in line 18.

The casts from interfaces to subtypes are analogous to the casts in the reverse direction. In Figure 4.16, since `y` has write access to the related memory, it is allowed to cast `y` to a write reference `z` in line 20. The borrowing context changes to $\{z \mapsto [z]\}$, as shown in line 21. Invoking `z.inc()` in line 22 is permitted since `z` possesses write access. However, `y.inc()` is prohibited because `y` lacks the access required by the method `inc`.

4.9 Recursive Struct

Inspired by Rust [7], to support linear recursive structures in the linear type system, we have introduced a concept similar to *Box* pointer, denoted as `*box` reference. Similarly, the size of a `*box` reference is fixed (equal to the size of a pointer, 8 bytes). However, we have added some other properties based on the uniqueness of our linear system. Firstly, a `*box` reference also embodies the concept of *Option* in Rust, which means that a `*box` reference could be `nil`. Secondly, a `*box` reference can only point to linear variables. Thirdly, when we create a struct instance with a `*box` reference pointing to a linear location, the ownership [1] of the linear location will be accumulated into the struct instance, and the stack of the linear location will be directly removed.

Currently, the `*box` reference is only used within the definition of a recursive struct, and we treat the `*box` reference as an array whose size can either be 0 (when the `*box` reference is `nil`) or 1 (when the `*box` reference is not `nil`). We can pass either a write reference with write access or a read reference with read access into a `*box` reference, and the ownership of the linear value associated with the reference will be accumulated into the struct that owns the `*box` reference.

Figure 4.17 is a small example of a recursive struct, we first declare a recursive struct named `node`, and line 5 creates a linear struct instance `n_1`, so we have a stack for `n_1` in the borrowing context, and the `next` field of `n_1` is `nil`, which is allowed due to the definition of `*box`. The borrowing context is $\{\&n\_1 \mapsto [\&n\_1]\}$. Then in line 7, we create another node `n_2`, and we specify that its next node is `n_1`, and the stack of `n_1` is removed from the borrowing context due to `n_2` obtaining full access to `n_1`, and the borrowing context changes to $\{\&n\_2 \mapsto [\&n\_2]\}$. Alternatively, we can view `node{2, &w n_1}` as a constructor function call, where the second argument is specified without the inout keyword.

```
1  type node struct {
2      value int
3      next *box node //box = option + write access + function call
4  }
5  n_1! := node{1,nil}; //ok, the box pointer can be nil
6  // {&n_1->[&n_1]}
7  n_2! := node{2,&w n_1}; // ok
8  // {&n_2->[&n_2]}, delete the stack of n_1 from X
```

Figure 4.17: Example code of using `*box`

4.10 Summary of the Extended Syntax

[1] Concept from Rust.

Syntax	Meaning
?	Linear read modifier
!	Linear write modifier
$\&r$	Create a read reference to linear variables
$\&w$	Create a write reference to linear variables
T	Type
$*r\ T$	Type of a read reference
$*w\ T$	Type of a write reference
$\&toS\ T$	Convert a linear write variable to a shared variable
$\&toL\ T$	Convert a shared variable to a linear write variable
$*box\ T$	Used in recursive structs, the type of a recursive field
$Interface < r >$	Type of an interface instance, having read access
$Interface < w >$	Type of an interface instance, having write access

Table 4.2: Syntax of the linear type system

Chapter 5

Case Study

This chapter presents some common use cases of our linear type system. For each use case, we propose a relevant Go example, and then verify the example using Gobra with and without our linear type system and compare the differences. Since we do not have a complete implementation of the designed linear type system, the linear Gobra examples show how we envision an implementation of linear Gobra to work.

5.1 Simple Example

We begin this section with a straightforward Gobra code snippet, demonstrating how to annotate programs in linear Gobra and how the borrowing context changes during the procedure. The following example shows a code snippet in Gobra without the linear type system. Line 8 declares a shared variable `a` and then passes the pointer to `a` as an argument of the function `incr`. Within function `incr`, `a` is increased by `n`. To modify `a`, the function `incr` is required to obtain the write permission to `a`, so we have the precondition `requires acc(a)`. After function `incr`'s execution, the permission of location `a` is returned by the postcondition.

```
 1  requires acc(a)
 2  ensures  acc(a)
 3  ensures  *a == old(*a) + n
 4  func incr(a *int, n int) () {
 5    *a += n
 6  }

 7  func client() () {
 8    a@ := 2
 9    n := 2
10    incr(&a, n)

11    assert a == 4
12  }
```

With the introduction of the linear type system, there is no longer a need to use permission reasoning for this code fragment, as shown below. In line 6, we merely need to declare `a` as a linear write variable. To grant function `incr` the access to modify `a`, we pass in a write reference to `a` into the function `incr`. Because the first parameter of the function `incr` is annotated with *inout*, the stack of `a` stays unchanged in the borrowing context after the function call.

The `incr` function does not have preconditions and postconditions related to permission reasoning, reducing the burden of annotations. While this thesis omits support for the linear system at the *encoding* level, it is clear that we no longer require permission reasoning in Viper files, which can also reduce the verification overhead.

```
 1  ensures  *a == old(*a) + n
 2  func incr(a inout *w int, n int) () {
 3    *a += n
 4  }

 5  func client() () {
 6    a! := 2
 7    // {&a -> [&a]}
 8    n := 2
 9    incr(&w a, n)
10    // {&a -> [&a]}

11    assert a == 4
12  }
```

5.2 Array and Slice

Since a slice can be viewed as a reference to an array, we address slices and arrays together in this section. To aid comprehension, we start with another simple example. We provide the entire annotated example in Gobra without the linear type system in Appendix A, which is from the evaluation examples of Gobra [9]. This example closely resembles the previous code in Section 5.1, with the only difference being that we change the parameter of the function `incr` from an integer reference to a slice.

From the perspective of the linear type system, there are no significant alterations. We simply declare `a` as a linear write array and create a slice that has write access to the location `a[2:]` and pass the slice into the function `incr`. Subsequently, the function `incr` increases each element within the given slice by `n`. The modifications to other postconditions and invariants are in fact unrelated to the characteristics of the linear type system.

It is worth noting that when we first enter the function `incr`, the borrowing context is $\{s \mapsto [s]\}$, as shown in line 3. In line 8, we attempt to modify `s[i]`, and rules `ReadLinearDereference` (4.1), `WriteLinearDereference`

(4.10), `ReadIndex` (4.9), and `WriteIndex` (4.15) are applied. First, the `WriteLinearDereference` rule (4.10) is applied to obtain a linear write array referenced by `s`. Subsequently, the `WriteIndex` rule (4.15) is applied to get write access to the element in the array. Therefore, we have the required write access to modify `s[i]`. Also, after each loop iteration, the borrowing context is $\{s \mapsto [s]\}$, shown in line 9. Therefore, the borrowing context is still $\{s \mapsto [s]\}$ after the whole loop procedure, shown in line 11.

```
1  ensures  forall k int :: 0 <= k && k < len(s) ==> s[k] == old(s[k]) + n
2  func incr (s inout w[]int, n int) {
3    // {s -> [s]}
4    invariant 0 <= i && i <= len(s)
5    invariant forall k int :: i <= k && k < len(s) ==> s[k] == old(s[k])
6    invariant forall k int :: 0 <= k && k < i ==> s[k] == old(s[k]) + n
7    for i := 0; i < len(s); i += 1 {
8      s[i] = s[i] + n // allowed, because s has write access
9      // {s -> [s]}
10   }
11   // {s -> [s]}
12 }

13 func client () {
14   a! := [4]int { 1, 2, 4, 8 }
15   // {&a -> [&a]}
16   incr(w a[2:], 2)
17   // {&a -> [&a]}

18   assert a[0] == 1 && a[1] == 2
19   assert a[2:][0] == a[2] && a[2:][1] == a[3]
20   assert a == [4]int { 1, 2, 6, 10 }
21 }
```

In Appendix A, Gobra utilizes quantified permissions in lines 4 and 5, which enable users to specify the permissions for each slice location. This is suitable for modeling heap structures that can be traversed in multiple directions, random-access data structures like arrays, and unordered data structures like graphs, for permission reasoning [8]. However, in linear Gobra, we do not need to reason permissions in such a way. From rules `ReadGetField` (4.8), `ReadIndex` (4.9), `WriteGetField` (4.14), and `WriteIndex` (4.15), we know that the access possessed by a struct or array propagates to all the fields of the struct or elements of the array.

Next, we examine a more complex example. Below is a piece of code that implements heap sort in linear Gobra. For the sake of clarity and readability, we have removed specifications unrelated to the linear type system and retained only the core code that implements the heap sort algorithm. Similarly, due to space constraints, for some excessively lengthy specifications, we will use pseudo-comments in natural language or delete them. For readers interested in the complete code and its detailed annotations, please refer to Gobra [9]-`heapsort.gobra` for the full version. This code may involve many Gobra features unrelated to this thesis, and readers can go to [8] for reference.

The function `parent` calculates and returns the index of the parent node of the heap element at the given index `i`. The function `leftChild` and the function `rightChild` calculate and return the index of the left child node and right child node of the heap element at the given index `i`, respectively. The function `swap` swaps the given two elements in the slice. The function `heapsort` is the main heapsort function. It starts by converting the input slice `s` into a max heap using the `heapify` function. Afterwards, the code repeatedly swaps the maximum element (at the root) with the last element in the heap to reduce the size of the unsorted heap. This process continues until the entire slice is sorted.

In line 13, the function `swap` receives a slice `s` and then modifies the slice, which is allowed because the slice has write access to the related memory, indicated by `w[]int`. The borrowing context remains unchanged during the execution of the function `swap`. Within function `heapsort`, we first pass `s` into the function `heapify` to construct a max heap, and according to the signature of the function `heapify`, in which `s` is annotated with inout, the borrowing context is still $\{s \mapsto [s]\}$ after the function call. Lastly, the function `heapsort` iteratively calls function `swap` to swap the root element of the heap and the last element of the slice and then calls `siftDown` to maintain the max heap. In these two callee functions, since the slices to be sorted are annotated with inout, we do not change the borrowing context after invoking these functions. And within function `siftDown`, there is a `if-else` branch, and because the borrowing contexts of the two branches are the same, namely $\{s \mapsto [s]\}$, the borrowing context after the merge is still $\{s \mapsto [s]\}$, as shown in line 64.

It is important to clarify that our linear type system does not perform boundary checks on arrays or slices. Therefore, while the code verification process does not need to inspect permissions, it is still imperative for the verification process to guarantee that all accesses remain within the bounds of the data structures.

```
1   pure func parent (i int) int {
2     return (i - 1) / 2
3   }
4   pure func leftChild (i int) int {
5     return 2 * i + 1
6   }
7   pure func rightChild (i int) int {
8     return 2 * i + 2
9   }

10  ensures old(s[i]) == s[j]
11  ensures old(s[j]) == s[i]
12  func swap(s inout w[]int, i, j int) {
13    // {s -> [s]}
14    tmp := s[i]
15    s[i] = s[j] // allowed, because s has write access
16    s[j] = tmp
17    // {s -> [s]}
18  }

19  ensures forall a, b int :: 0 <= a && a <= b && b < len(s) ==> s[a]<=s[b]
20  func heapsort (s inout w[]int) {
21    // {s -> [s]}
22    heapify(s)
23    // after function call: {s -> [s]}
24    end := len(s) - 1

25    for 0 < end {
26      swap(s, 0, end)
27      // after function call: {s -> [s]}
28      end = end - 1
29      siftDown(s, 0, end)
30      // after function call: {s -> [s]}
31    }
32    // {s -> [s]}
33  }

34  ensures forall n int::0<=parent(n)&&n<len(s)==>s[n]<=s[parent(n)]
35  func heapify (s inout w[]int) {
36    // {s -> [s]}
37    start := parent(len(s) - 1)
38    for 0 <= start {
39      siftDown(s, start, len(s)-1)
40      // after function call: {s -> [s]}
41      start = start - 1
42      // {s -> [s]}
43    }
44    // {s -> [s]}
45  }

46  ensures forall i int::0<=i&&i<=end&&end+1<len(s)==>s[i]<=s[end+1]
47  func siftDown (s inout w[]int, start int, end int) {
48    // {s -> [s]}
49    root := start
50    stop := false

51    for !stop && leftChild(root) <= end {
52      child := leftChild(root)
53      if (child + 1 <= end && s[child] < s[child + 1]) {
54        child = child + 1
55      }
56      if (s[root] < s[child]) {
57        swap(s, root, child)
58        // after function call: {s -> [s]}
59        root = child
60      } else {
61        stop = true
62        // {s -> [s]}
63      }
64      // after merge: {s -> [s]}
65    }
66    // {s -> [s]}
67  }

68  func client1 () {
69      arr! := [6]int{12, 11, 13, 5, 6, 7}
70      fmt.Println("Unsorted␣array:", arr)
71      // before the function call: {&arr -> [&arr]}
72      heapsort(w arr[:])
73      // after the function call: {&arr -> [&arr]}
74      fmt.Println("Sorted␣array:", arr)
75  }

76  func client2 () {
77      s := w[]int{12, 11, 13, 5, 6, 7}
78      fmt.Println("Unsorted␣slice:", s)
79      // before the function call: {s -> [s]}
80      heapsort(s)
81      // after the function slice: {s -> [s]}
82      fmt.Println("Sorted␣array:", s)
83  }
```

We have two client functions. In the `client1` function, a linear write array `arr` is declared in line 69. In line 72, `arr` is sliced using `[:]`, and write access is granted to the slice using `w`. Subsequently, the anonymous slice is passed into the function `heapsort`. The borrowing context before and after the function call is described in lines 71 and 73, respectively. In the `client2` function, we declare a slice `s` and indicate that `s` has write access to the memory with `w[]int`. Consequently, we can pass `s` as an argument of the function `heapsort`.

5.3 Struct

Due to the introduction of the linear type system, there are certain distinctions when dealing with structs with recursive references, as we introduced the `*box` reference. Consequently, we divide this chapter into two parts, Structs without recursive references and Structs with recursive references, to demonstrate the use of the linear type system in different scenarios.

5.3.1 Struct without Recursive Reference

In structs, there are typically many fields, so we often use *predicates* for permission reasoning, which gives a name to a parameterized assertion. A predicate can have any number of parameters, and its body can be any self-framing Gobra assertion using only these parameters as variable names [8]. Predicate definitions can be recursive, allowing them to denote permission to and properties of recursive heap structures such as linked lists and trees, which we will see in the next subsection.

The following snippet is an example of a predicate. Lines 1 to 3 define the structure of type `DenseMatrix`. The type `DenseMatrix` has a field named `values`, which is a reference to a two-dimensional array. Lines 4 to 9 define a predicate named `denseMatrix`, which is used to obtain the permissions of the given reference `m`.

```
1   type DenseMatrix struct {
2       values *[100][100]int
3   }

4   pred denseMatrix(m *DenseMatrix) {
5       acc(&m.values) &&
6       (forall i int::(0 <= i <len(m.values) ==> acc(&m.values[i])))&&
7       (forall i, j int :: (0 <= i < len(m.values) &&
8       0 <= j < len(m.values[i]) ==> acc(&m.values[i][j])))
9   }

10  // number of rows
11  requires acc(denseMatrix(m), _)
12  pure func (m *DenseMatrix) lenX() int {
13      // return len(m.values): not allowed
14      return unfolding acc(denseMatrix(m), _) in len(m.values)
15  }
```

In Gobra, predicate instances are not equivalent to their body [8]. For instance, we use the `denseMatrix` as a precondition for the method `lenX`, and it is not allowed to directly access `m.values`, as shown in line 13. Therefore, Gobra has `fold` and `unfold` annotations to handle this. To make the snippet verify, Gobra requires an additional unfold annotation, instructing Gobra to replace a predicate instance with its body. Conversely, the fold annotation exchanges the predicate body with the predicate instance. For instance in line 14, we first `unfold acc(denseMatrix(m), _)` and then it is allowed to access `m.values`.

Next, we continue to use the struct type `DenseMatrix` as the example. Similarly, due to space limitations, some non-essential parts have been omitted. We have defined several methods the type `DenseMatrix`: the method `lenX` returns the number of rows in the matrix, the method `lenY` returns the number of columns in the matrix, and the `lookup` method returns the value at row `i` and column `j` in the matrix. These methods do not perform any modifications to the given `DenseMatrix`, so their receivers are read references to `DenseMatrix`. Within these functions, we need to read from `m.values`. Since `m.values` is a short-hand for `(*m).values`, we need to apply `ReadLinearDereference` (4.1) and `ReadGetField` (4.8) rules to obtain read access to `(*m).values`. Thus, reading from `m.values` is allowed by the linear system and we do not need `fold` and `unfold` functions to obtain access to the field of a struct. The last function is `multAlt`, which multiplies every element in the matrix by a factor of `z`. This operation requires write access to the receiver matrix. Therefore, the receiver needs to be a write reference to `DenseMatrix`.

Through comparison, it can be observed that by employing the linear type system, we can avoid the need for the predicate `denseMatrix` and `fold` and `unfold` annotations in some cases. From the perspective of a programmer, this approach significantly reduces the complexity and number of annotations required in programs. Meanwhile, the verification overhead is also reduced.

```
type DenseMatrix struct {
    values *w[100][100]int
}

// number of rows
pure func (m inout *r DenseMatrix) lenX() int {
    return len(m.values)
}

// number of columns
requires m.lenX() != 0
pure func (m inout *r DenseMatrix) lenY() int {
    return len((m.values)[0])
}

requires 0 <= i && i < m.lenX()
requires 0 <= j && j < m.lenY()
pure func (m inout *r DenseMatrix) lookup(i, j int) (res int) {
    return (m.values)[i][j]
}

requires m.lenX() != 0
ensures m.lenX() == old(m.lenX()) && m.lenY() == old(m.lenY())
ensures forall i, j int::(0 <= i <m.lenX() && 0 <= j <m.lenY()
        ==> m.lookup(i, j) == old(m.lookup(i, j)) * z)
func (m inout *w DenseMatrix) multAlt(z int) {
    // {m -> [m]}
    // invariants have been omitted
```

```
25      for x := 0; x < m.lenX(); x++ {
26          // invariants have been omitted
27          for y := 0; y < m.lenY(); y++ {
28              (m.values)[x][y] = (m.values)[x][y] * z
29              // {m -> [m]}
30          }
31      }
32      // {m -> [m]}
33  }

34  func client () {
35      var matrix *w[100][100]int = &w ...
36      // {matrix -> [matrix]}
37      call(matrix)
38      // {matrix -> [matrix]}
39      // All entries in the matrix have been multiplied by 2
40      matrix[0][0] = 3 // can still modify matrix
41  }

42  func call (values inout *w[100][100]int) {
43      // {values -> [values]}
44      dm := &w DenseMatrix{values}
45      // {values -> [*dm.values # values], [dm -> [dm]}
46      dm.multAlt(2)
47      // {values -> [*dm.values # values], [dm -> [dm]}
48  }
```

We also have a `client` function showing how to manipulate the type `DenseMatrix`. In line 35, we declare a write reference `matrix` to a two-dimensional array. The borrowing context is $\{matrix \mapsto [matrix]\}$. Then the function `call` is invoked in line 37. Since the parameter `values` of the function `call` is annotated with inout, the borrowing context stays unchanged after this call. When entering the function `call`, the borrowing context is $\{values \mapsto [values]\}$, as shown in line 43. Line 44 declares a write reference `dm` pointing to a `DenseMatrix` instance, which takes `values` as the field. Thus, the borrowing stack of `values` changes to $values \mapsto [*dm.values \# values]$, as shown in line 45.

5.3.2 Struct with Recursive Reference

As we discussed in Section 4.9, we have introduced a new `*box` reference. The following is an example of using `*box` references. First, we define the structure of `Tree`, which contains three fields: `Left`, `Value`, and `Right`. `Left` and `Right` are `*box` references to other `Tree` nodes, and `Value` is an integer value. The method `Contains` checks if a given value `v` exists in the binary tree.

The `Insert` method inserts a new node with value `v` into the binary tree. The method decides whether to insert the new node as the left child or right child based on whether the sum of the current node's value and `v` is even or odd. Since the write reference and read reference cannot be `nil`, the precondition of the method `Insert` requires that `self` is not `nil`. Because `self` is a write reference to `Tree`, the method `Insert` is allowed to modify `self`. When entering the `Insert` method, the borrowing context is

$\{self \mapsto [self]\}$, which stays the same within the entire function. Notice that the receiver `self` is not annotated with inout, but we return `self` as the return value. Thus, `self` is still accessible after the `Insert` method. However, the frame information in the borrowing stack of the receiver before invoking the `Insert` method will be lost.

```
type Tree struct {
    Left  *box Tree
    Value int
    Right *box Tree
}

requires self != nil
pure func (self inout *r Tree) Contains(v int) bool {
    return self != nil && self.Value == v ||
    (self.Left != nil && self.Left.Contains(v)) ||
    (self.Left != nil && self.Right.Contains(v))
}

requires self != nil
ensures res.Contains(v)
func (self *w Tree) Insert(v int) (res *w Tree) {
    // {self -> [self]}
    if (self.Value + v) \% 2 == 0 {
        if self.Left == nil {
            self.Left = &w(Tree{Value: v})
            // {self -> [self]}
        }else{
            self.Left = self.Left.Insert(v)
            // {self -> [self]}
        }
        // after merge: {self -> [self]}
    } else {
        if self.Right == nil {
            self.Right = &w(Tree{Value: v})
            // {self -> [self]}
        }else{
            self.Right = self.Right.Insert(v)
            // {self -> [self]}
        }
        // after merge: {self -> [self]}
    }
    // after merge: {self -> [self]}
    return self
}
```

In practice, the restriction that `*box` references can only be used when defining a struct has certain limitations. For example, since we constrain that the write and read references that already exist in the borrowing context cannot be `nil` and the `*box` reference can only be used within the definition of a struct, the linear type system cannot support the method `DeleteAll` because the `DeleteAll` method may produce a `nil` value (all nodes in the tree are deleted).

We will explore how our system would support the method `DeleteAll` if we allow to use `*box` references outside the definition of a struct. The method `DeleteAll` removes all nodes with the value `v` from the binary tree. The `deleteLeftMost` method is a helper function for the method `DeleteAll`, which deletes the leftmost node in a tree.

```
requires self != nil
ensures res != nil ==> !res.Contains(v)
func (self *w Tree) DeleteAll(v int) (res *box<w> Tree) {
    // {self -> [self]}
    var newLeft, newRight *box Tree
```

```
 6      if self.Left != nil {
 7          newLeft = self.Left.DeleteAll(v)
 8      }
 9      if self.Right != nil {
10          newRight = self.Right.DeleteAll(v)
11      }
12      if self.Value == v {
13          if newLeft == nil {
14              // at most one subtree, thus use other one:
15              res = newRight
16          } else if newRight == nil {
17              // at most one subtree, thus use other one:
18              res = newLeft
19          } else {
20              // newRight != nil, newRight != nil
21              var leftMost int
22              self.Right, leftMost = newRight.deleteLeftMost(v)
23              // overwrite value that should be deleted:
24              self.Value = leftMost
25              self.Left = newLeft
26              res = self
27          }
28      }else {
29          newLeft
29          self.Left = newLeft
30          self.Right = newRight
31          res = self
32      }
33      return res
34  }
35  requires self != nil
36  requires !self.Contains(v)
37  ensures res != nil ==> !res.Contains(v) && v != leftMost
38  func (self *w Tree) deleteLeftMost(ghost v int)(res *box<w> Tree,
39                                                  leftMost int){
40      if (self.Left != nil) {
41          self.Left, leftMost = self.Left.deleteLeftMost(v)
42          res = self
43      } else if (self.Right != nil) {
44          res, leftMost = self.Right, self.Value
45      } else {
46          res, leftMost = nil, self.Value
47      }
48      return res, leftMost
49  }
```

In the method `DeleteAll`, we recursively traverse the tree looking for nodes with values equal to `v` and delete them. In line 5, we declare `newLeft` and `newRight` as `*box Tree`, representing the new subtrees of `self`, where all nodes are not equal to `v`. As shown in line 3 and line 38, the return values of the method `DeleteAll` and the method `deleteLeftMost` are of type `*box<w> Tree`, which are a `*box` reference with write access to a `Tree`. We do not go into details of these methods.

By allowing the use of `*box` references outside type definitions, we can provide appropriate signatures for the method `DeleteAll` and the method `deleteLeftMost`. However, at certain program points, it may be difficult to determine what the borrowing context is. For example, in lines 7 and 9, because the return values of `self.Left.DeleteAll(v)` and `self.Right.DeleteAll(v)` are of type `*box<w> tree`, it is not possible to determine if `newLeft` and `newRight` are nil or not. If `newLeft` and `newRight` are not nil, we should add their borrowing stacks into the borrowing context. Otherwise, there is no need to add the borrowing stacks of `newLeft` and `newRight`. Since a `*box` reference can be either `nil` or a read/write reference, the borrowing stack of a *box reference is uncertain. Due to our lack of research on *box references, we adhere to the design choice of only allowing the use of *box references when defining structures.

5.4 Interface

The following is a simplified example from Gobra [9]-visitor_pattern.gobra, which shows how our type system works with interfaces.

```
// visitor
type visitor interface {
  pure visitVar(v inout *r variable) int
  pure visitCons(c inout *r constant) int
}

type node interface {
  requires v != nil
  pure accept(v visitor<r>) int
}

// variable
type variable struct {
  id int
}
requires v != nil
pure func (self inout *r variable) accept(v visitor<r>) int {
  // {self -> [{self}], v -> [{v}]}
  return v.visitVar(self)
}

// constant
type constant struct {
  value int
}
requires v != nil
pure func (self inout *r constant) accept(v visitor<r>) int {
  // {self -> [{self}], v -> [{v}]}
  return v.visitCons(self)
}

// evaluator
type evaluator struct {
    f int
}
pure func (self inout *r evaluator) visitVar(v inout *r variable) int {
  // {self -> [{self}], v -> [{v}]}
  return v.id
}
pure func (self inout *r evaluator) visitCons(c inout *r constant) int {
  // {self -> [{self}], c -> [{c}]}
  return c.value
}

func evaluator_client() {
  ev := &r evaluator{f:2}
  // {ev -> [{ev}]}
  a := &r constant{42} // type of a: *r constant
  // {ev -> [{ev}], a -> [{a}]}
  res := ev.visitAddition(a)
  // {ev -> [{ev}], a -> [{a}]}

  var b node<r> = toInterface(&r variable{0})
  // {ev -> [{ev}], a -> [{a}], b -> [{b}]}
  res := ev.visitAddition(b)
  // {ev -> [{ev}], a -> [{a}], b -> [{b}]}

  c := b.(*r variable)
  // {ev -> [{ev}], a -> [{a}], c -> [{c}]}
}
```

From line 2 to line 5, we declare an interface named `visitor`. The interface defines two methods to visit two datatypes: `variable` and `constant`. These two datatypes are the subtypes of interface `node`. Lines 14 to 18 define the `accept` method for the type `variable` and line 16 shows the borrowing context within the method, which is $\{self \mapsto [\{self\}], v \mapsto [\{v\}]\}$. The method for type `constant` is analogous to the method for type `variable`.

Lines 20 to 22 declare the type `evaluator`, which is a subtype of interface `visitor`. The type `evaluator` implements method `visitVar` from line 32 to line 35. For the method, the borrowing context is $\{self \mapsto [\{self\}], v \mapsto [\{v\}]\}$, shown in line 33. The `visitCons` method is analogous.

We also have a `evaluator_client` function, line 43 creates a read reference `a` to `constant`. The type of `a` is `*r constant`. Line 45 invokes method `ev.visitAddition(a)`. Because the receiver and parameter are both annotated with inout, the borrowing context stays unchanged, which is $\{ev \mapsto [\{ev\}], a \mapsto [\{a\}]\}$. Then in line 47, we create a read reference to `variable` and then cast it to `node`. As discussed in Section 4.8, the target of this assignment, the variable `b`, is expected to possess the same access as the reference being cast. Therefore, the borrowing context changes to $\{ev \mapsto [\{ev\}], a \mapsto [\{a\}], b \mapsto [\{b\}]\}$, and stays the same after the function call in line 50. Lastly, line 51 casts `b` to a read reference to `variable` and assigns the result to `c`. Thus, the borrowing stack associated with `b` is eliminated from the borrowing context, and the borrowing stack of `c` is added to the borrowing context. The borrowing context changes to $\{ev \mapsto [\{ev\}], a \mapsto [\{a\}], c \mapsto [\{c\}]\}$.

5.5 Map

A map is similar to a slice, which is also a reference-type data structure. Below is an example that shows how our system supports maps.

The parameters of the function `getSalary` are a string `e` and a variable `s`, which holds read access (indicated by `r map[string]int`) to the memory location of a map. The borrowing context is $\{s \mapsto [\{s\}]\}$ within the function `getSalary`. Similarly, the function `modifySalary` takes three parameters: a string `e`, a variable `s`, which is a write reference (indicated by `w map[string]int`) to a map, and an int `salary`. Since `s` has write access to the memory location, updating `s` in the function `modifySalary` is allowed. Within the function `main`, we declare a variable `employeeSalaries`, which has write access to the memory location storing the map. The borrowing context is $\{employeeSalaries \mapsto [employeeSalaries]\}$. In line 26, we declare a variable `e2`, which has only read access to the same memory location. Therefore, the borrowing context changes to $\{employeeSalaries \mapsto [\{e2, employeeSalaries\} \ \# \ employeeSalaries]\}$, as shown in line 27.

```
1  func getSalary(e string, s inout r map[string]int) int {
2      // {s -> [{s}]}
3      salary, exists := s[e]
4      if exists {
5          return salary
6          // {s -> [{s}]}
7      }
8      // {s -> [{s}]}
9      return 0
10 }

11 func modifySalary(e string, s inout w map[string]int, salary int) bool {
12     // {s -> [s]}
13     oldSalary, exists := s[e]
14     if exists {
15         s[e] = salary
16         return true
17         // {s -> [s]}
18     }
19     // {s -> [s]}
20     return false
21 }

22 func main() {
23     var employeeSalaries w map[string]int
24     employeeSalaries = make(map[string]int)
25     // {employeeSalaries -> [employeeSalaries]}

26     var e2 r map[string]int = employeeSalaries
27     // {employeeSalaries -> [{e2, employeeSalaries} # employeeSalaries]}
28 }
```

5.6 Limitations of the Linear Type System

Although with the help of the linear type system, we can save the overhead of permissions reasoning, in some cases our system may demonstrate some limitations. Below is such an example and we provide codes for both Gobra with and without the linear type system respectively.

In line 1, we declare two linear write variables `x` and `y`, line 2 declares a write reference `p`, and line 3 declares a read reference `q`. The borrowing context is $\{\&x \mapsto [\&x], \&y \mapsto [\&y]\}$. After the `if-else` control flow, the borrowing stack is $\{\&x \mapsto [\{p, q\} \ \# \ p \ \# \ \&x], \&y \mapsto [\{p, q\} \ \# \ p \ \# \ \&y]\}$. If we want to modify the memory location that `p` points to, we need to `write_activate p` in all borrowing stacks. Afterwards, if we want to use `q` after `write_activate p`, the linear type system will report an error since `q` is deleted from the borrowing context.

```
1  x! := 5; y! := 6
2  var p *w int
3  var q *r int
4  // {&x -> [&x], &y -> [&y]}

5  if some_condition_known_at_runtime {
6      p = &w x
7      // {&x -> [p # &x], &y -> [&y]}
8      q = &r (*p)
9      // {&x -> [{p,q} # p # &x], &y -> [&y]}
} else {
    p = &w y
    // {&x -> [&x], &y -> [p # &y]}
    q = &r (*p)
    // {&x -> [&x], &y -> [{p,q} # p # &y]}
}
// after merge: {&x -> [{p,q} # p # &x], &y -> [{p,q} # p # &y]}
*p=4;
// {&x -> [p # &x], &y -> [p # &y]}
print!("{}",p) // allowed
print!{"{}",q} // not allowed, q is not in the borrowing context
```

The following shows the example in Gobra without linearity. Line 1 declares two shared variables, `x` and `y`, and line 2 declares two references `p` and `q`. If the `if` branch is taken, then reference `p` and `q` will point to `x` otherwise they will point to `y`. With permission reasoning, all accesses after line 10 are allowed.

```
1   x@ := 5; y@ := 6
2   var p,q *int
3   if some_condition_known_at_runtime {
4       p = &x
5       q = &(*p)
6   } else {
7       p = &y
8       q = &(*p)
9   }
10  *p=4;
11  print!("{}",p) // allowed
12  print!{"{}",q} // allowed
```

The example above illustrates that our designed linear type system might be overly restrictive, and will potentially rule out correct codes in some cases.

Another limitation is that in the designed linear type system, the accesses associated with read references and write references cannot be altered. If a reference is declared as a read reference, it is impossible to modify the memory through this reference. Whereas in Gobra without linearity, we can use fractional permission to allocate different permissions to a reference. For example, if we have a reference `x`, we can assign read permission to `x` using `acc(x, _)`. Subsequently, if we want to grant `x` write permission, we can use `acc(x)` to achieve this.

5.7 Summary

From the previous analysis, it is evident that our linear type system is suitable for complex data types having recursive fields or hierarchical structures that require permission reasoning. We can get rid of the annotation overhead of memory reasoning including predicates and `fold` and `unfold` annotations. The transfer of permissions between functions or methods is also not needed because the linear type system takes care of the permission-related concerns.

Due to the reference management mechanisms in the linear type system, references may automatically become inaccessible due to the changes in the borrowing context. If programmers do not want a reference to become invalid, or if they want to manually manage the permissions associated with a reference, it is more appropriate to use Gobra without linearity.

Chapter 6

Conclusion

To simplify the memory reasoning for heap-allocated values, we design a linear type system for Gobra. We add two type modifiers, linear read and linear write, to Gobra's type system to introduce linearity into Gobra. Also, we introduce read and write references for linear values. The basic idea of our linear type system is that multiple read-only references may exist at the same time, but write references cannot exist with other references. Thus, the type system tracks references to ensure linearity and takes the responsibility to protect memory safety. We introduce a context in our type judgments, namely the borrowing context. This allows the type system to determine whether a reference or variable has permission to perform operations. For instance, if we intend to modify a memory location through a reference, the type system mandates that the reference must have write access to that memory location. Furthermore, we developed the type system's support for features like conversion, function calls, and recursive structures involving linear references.

In Chapter 5, we demonstrate how to replace permission reasoning using the linear type system, which shows that the annotation overhead has decreased and our design is suitable for complex structs having recursive fields or hierarchical structures. While the lack of an implementation prevents us from obtaining concrete measurements, we are highly optimistic that the overall verification time could decrease substantially if Gobra supported such a type system. Then, we propose prospective avenues for future research.

6.1 Future Work

Explore the possibilities and limitations of using graphs for the borrowing context. Currently, we employ a set of stacks to implement the borrowing context, facilitating the tracking of linearity and reference information. Nevertheless, after conducting preliminary experiments, it appears that Directed Acyclic Graphs (DAGs) possess a structural advantage for serving as the foundational structure for the borrowing context. However, without in-depth investigation, we conservatively estimate that the code scope supported by stacks and DAGs does not exhibit significant disparities. Moreover, opting for a DAG necessitates ensuring the absence of cyclic structures within the entire graph, potentially introducing complexities in node insertion, deletion, or establishing inter-node edges. Consequently, this paper continues to utilize the stack-based structure for the borrowing context. We have not conducted an exhaustive exploration of the possibilities and limitations of using graphs, which should be studied intensively in the future.

Investigate the compatibility of the linear type system with concurrency. As mentioned before, Go is a popular programming language known for its simplicity and concurrency support, and goroutines are a feature in Go for implementing concurrency. Gobra already supports the creation of goroutines using regular functions or method calls [1]. In theory, based on our current design of the linear type system, we do not need to introduce new features to support data structures like channels, which is similar to map. However, channels are typically used in concurrent scenarios, and the compatibility of our linear type system with concurrency is still unknown. Therefore, further research is needed in this area.

Implement the type system in Gobra. We do not provide an implementation currently. A complete implementation would be helpful for analyzing the expressiveness and efficiency of the designed type system in the future. Additionally, in a naive implementation, certain operations on the borrowing context can lead to the loss of linearity information. For instance, when we convert a linear value into a non-linear value, we directly remove the borrowing stack associated with the linear location, leading to a loss of information about the frames present in the borrowing stack. Therefore, it is advisable to contemplate if there are alternative and superior methods of managing information in the borrowing context during the implementation.

Extend the linear type system to support closures. This thesis has focused exclusively on regular function calls and method invocations, without delving into the provision of support for closures. Consequently, the next step involves extending the linear type system to encompass closures.

Evaluate the linear type system on larger Go programs. We aspire to apply our linear type system to larger Go programs, including projects like VerifiedSCION [10] and WireGuard [11]. This endeavor will provide us with a more comprehensive evaluation of the linear type system's effectiveness.

Bibliography

[1] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs (extended version)," *CoRR*, vol. abs/2105.13840, 2021. [Online]. Available: https://arxiv.org/abs/2105.13840

[2] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theoretical Computer Science*, vol. 375, no. 1, pp. 271–307, 2007, festschrift for John C. Reynolds's 70th birthday. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S030439750600925X

[3] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*, 1990.

[4] X. Denis, J.-H. Jourdan, and C. Marché, "Creusot: A foundry for the deductive verification of rust programs," in *Formal Methods and Software Engineering*, A. Riesco and M. Zhang, Eds. Cham: Springer International Publishing, 2022, pp. 90–105.

[5] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust (invited)," in *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

[6] J. Li, A. Lattuada, Y. Zhou, J. Cameron, J. Howell, B. Parno, and C. Hawblitzel, "Linear types for large-scale systems verification," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527313

[7] R. Group, "Rust tutorials," https://doc.rust-lang.org/stable/book/ch04-00-understanding-ownership.html.

[8] G. Group, "Gobra tutorials," https://github.com/viperproject/gobra/blob/master/docs/tutorial.md.

[9] G. R. Group, "Gobra evaluations," https://github.com/viperproject/gobra/tree/d2acefbbdb9d8f7fcdf14455319d8c79b241f570/src/test/resources/regressions/examples/evaluation.

[10] N. S. Group and the Information Security Group, “The verifiedscion project,” https://www.pm.inf.ethz.ch/research/verifiedscion.html.

[11] J. A. Donenfeld, “Wireguard: Next generation kernel network tunnel,” in *Network and Distributed System Security Symposium*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:2590070

Appendix A

Complete Code Covered in Section 5.2

```
// Any copyright is dedicated to the Public Domain.
// http://creativecommons.org/publicdomain/zero/1.0/

package pkg

requires forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])
ensures  forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])
ensures  forall k int :: 0 <= k && k < len(s) ==> s[k] == old(s[k]) + n
func incr (s []int, n int) {
  invariant 0 <= i && i <= len(s)
  invariant forall k int :: 0 <= k && k < len(s) ==> acc(&s[k])
  invariant forall k int :: i <= k && k < len(s) ==> s[k] == old(s[k])
  invariant forall k int :: 0 <= k && k < i ==> s[k] == old(s[k]) + n
  for i := 0; i < len(s); i += 1 {
    s[i] = s[i] + n
  }
}

func client () {
  a@ := [4]int { 1, 2, 4, 8 }
  incr(a[2:], 2)

  assert a[0] == 1 && a[1] == 2
  assert a[2:][0] == a[2] && a[2:][1] == a[3]
  assert a == [4]int { 1, 2, 6, 10 }
}
```

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A Linear Type System for Gobra

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Han	Liming

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 2023-10-01	Han Liming 韩黎明

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.