Linear Type System for Gobra

Master's Thesis Project Description

Liming Han

Supervisors: Felix Wolf, João Pereira, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zurich

Start: 27th February 2023

1 Introduction

Go is a popular programming language known for its simplicity, concurrency support, and efficient execution. 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 takes Go programs annotated with specifications and proof annotations and returns true if and only if the Go 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 accesses a memory location, it is required to hold the associated permission [1]. The permission to modify a location `v` is denoted in Gobra by `acc(&v)`. Permissions are held by method executions and transferred between methods upon call and return [1].

Figure 1 shows an annotated Go program with two functions using permission reasoning: `foo` and `add5`. In function `foo`, we declare an int variable `x` which is heap-allocated. Function `add5`'s parameter `y` is a pointer to `x`. The permission to modify the pointer `y` is obtained by the method `add5` via the precondition `requires acc(y)`. Furthermore, the permission is transferred back to the caller via the postcondition `ensures acc(y)`. Because the method `add5` has acquired the permission to `y` through its precondition, the method is allowed to update `y` in its body.

However, permission reasoning is expensive. In particular, permission reasoning causes 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 non-termination of the verification process. Rust verifiers (e.g., Creusot [3] and Prusti [4]) have illustrated that linear type systems can be used to simplify heap reasoning. In addition, Linear Dafny [5] also has shown that by using linear types, one can simplify the proof obligations ultimately passed to the backend, which results in better verification performance. According to Linear Dafny [5], the linearized VeriBetrKV has 28% fewer lines of proofs, and 30%

```go
func foo() {
    var x@ int = 42
    var y *int = &x
    add5(y)
}
requires acc(y)
ensures acc(y)
func add5(y *int) {
    *y += 5
}
```

Figure 1: Example code for using permission reasoning in Gobra

```
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 2: No duplication and no discard

```
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = length(s1);
}

fn length(s: String) -> (String, usize) {
    let len = s.len();
    (s, length)
}
```

Figure 3: Rust code without using borrowing

shorter verification time overall. Both papers demonstrate the effectiveness of combining verification with linear types.

Inspired by these two papers, the aim of this thesis is to simplify memory reasoning by introducing a linear type system in Gobra. To be more specific, we aim to study the viability of introducing a linear type system to Gobra and its effectiveness in simplifying the specification and verification of Go programs.

2 Context

The (strict) linear type has *no duplication* and *no discard* properties [6]. *No duplication* guarantees that, at any moment, there is at most one reference per memory location. Because of this, there is no aliasing. This makes reasoning about heap-allocated data structures simpler and allows the compiler to perform more aggressive optimizations. For example, Figure 2 shows a Go program with 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 (strict) linear type systems because of the no duplication property. *No discard* means that programmers should release memory explicitly, as shown in Line 5 of Figure 2, which avoids the overhead of garbage collection. These two features ensure that the linear type system can satisfy data safety.

Generally, current linear type systems have richer features, such as the introduction of *borrowing* in Rust [7]. In Rust, every variable has a method that conceptually owns the variable, the *owner*. A variable can be used only if it is in the scope of its owner. The ownership of a variable is transferred between functions upon parameters and return values. For example in Figure 3, we have a string variable `s1` declared in `main` function, therefore the owner of `s1` is function `main`. When `main` invokes `length`, the ownership of variable `s1` is moved to function `length`. If we still want to use this string variable in `main` afterwards, we need to return the ownership by returning the value as an output parameter.

Borrowing is the action of creating a reference that allows one to refer to some value without taking ownership of it. As can be seen in Figure 4, we can use `s1` without returning `s1`.

Figure 5 implements the same functionality as in Figure 1, and we use this code snippet to illustrate how a linear type system that combines the above properties may conceptionally look like in Go and how we can get rid of permission reasoning by using such a linear type system. We use `x!` to indicate that variable `x` is a linear type, and `y` is the *borrowing* of `x`. Inside the body of `add5`, the access to `y` is safe because the linear type system guarantees that no other access to `y` happens simultaneously.

```
fn main() {
    let s1 = String::from("hello");
    let len = length(&s1);
}

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

Figure 4: Rust code using borrowing

```
func foo() {
    var x! int = 42
    var y *int = &x
    add5(y)
}

func add5(y! *int) {
    *y += 5 // safely update
}
```

Figure 5: Example code for using linear type systems in Go

2.1 Design Space

For this thesis, the main goal is to design a linear type system extension for Gobra. However, there is a large design space for such linear type systems. That includes many concepts and tradeoffs, such as:

- **Borrowing.** Rust offers expressive borrowing mechanisms. Rust distinguishes between *mutable borrowing* and *immutable borrowing*. *Immutable borrowing* means that a method holds temporary read-only permission to the heap location and can not modify the location. *Mutable borrowing* means that a method holds temporary write permission to a heap location, therefore the method can update the location. We can have multiple immutable borrowings of a variable at the same time since that immutable borrowing requires read-only permission to linear variables [6]. Conversely, the mutable borrowing of a variable must be exclusive because the write permission to a variable must be unique [6]. Introducing support for mutable borrowing and immutable borrowing can enhance the expressiveness of the type system. However, this may also result in increased complexity of the linear type system.

- **Complexity and Performance.** As mentioned before, Linear Dafny and Rust are both languages with linear type systems. The type system for Rust is more sophisticated and can handle a lot of code with concepts such as borrowing. Conversely, the type system for Linear Dafny can handle less code but is less complicated, and thus is easier to use.

  There is a tradeoff: we can get the same coverage of programs either by making the type system more sophisticated and having less to no verification obligation or by making the type system less sophisticated, but having more verification obligations. For example, we can use permission reasoning to support a linear type system. Figure 6 illustrates why permission reasoning can be useful in a linear type system. Line 3 assigns a nonlinear type `x` to a linear type `y`, which is usually not allowed. However, the precondition `requires acc(x)` implies that other methods do not have permission to read or modify the pointer `x`. Therefore, we know that other methods cannot modify or read `x`. Furthermore, since the method `foo` does not return the permission to `x`, other methods cannot access `x` after the execution. In other words, even though there might be aliases of `x`, these aliases are not usable. As a consequence, the conversion from nonlinear to linear in Line 3 is justified, if at the assignment, Gobra removes all permissions to the converted memory location. It is not enough to just check that the method execution holds all permissions, because other aliases may be made usable again once the permissions are transferred back after method `foo` returns. In conclusion, the interaction between linear and non-linear is an important aspect of the design space. The aforementioned conversions are one potential approach.

```
requires acc(x)
func foo(x *int) {
    var y linear *int = x // usually does not hold
}
```

Figure 6: Conversion between the linear type system and permission reasoning

- **Affinity vs. strictly linearity.** Existing linear type systems make a distinction between *strictly linear type systems* and *affine type systems* [6]. In strictly linear type systems, every linear variable must be used exactly once and needs to be released explicitly. In affine type systems, a linear variable can be used at most once and does not need to be discarded manually.

  From the perspective of program verification, if we choose strictly linearity, Gobra has to enforce that all linear variables eventually be released explicitly. However, this check is not required in an affine type system.

  For this design space, we should study whether we get any benefits from the stronger guarantees of a strictly linear type system that are worth the loss of expressiveness caused by the additional restrictions of a strictly linear type system.

3 Core Goals

Identifying the requirements for a linear type system in Go. The aim of this goal is to identify requirements that a linear type system extension must satisfy to be applicable to Go. For each of the design choices outlined in Section 2.1, we will investigate to which extent the design choices can be applied to Go. In particular, we will analyze the advantages and disadvantages of each design choice. To achieve this goal, we will conduct a manual analysis of relevant real-world Go code, such as Go's Wireguard and Scion implementations. Furthermore, our analysis might leverage Go's existing static analysis framework, which for instance offers an automated may-alias analysis, to cover a larger collection of Go programs.

Design a linear type system for a small subset of Go. As the second goal, we will use the results from the first goal to design a linear type system for Gobra. For this goal, we will target a small subset of Go without closures, interfaces, channels, arrays, slices, and maps. We will use existing works as inspiration for our type system, such as Linear Dafny [5], Rust [7], and the linear type system by Wadler [6].

Add support for interfaces. As the third core goal, we extend the type system to further support Go's interfaces.

Add support for arrays, slices, maps, and channels. As the fourth core goal, we extend the type system to further support arrays, slices, maps, and channels.

Implementation. We plan to implement a prototype of the type system in Gobra. In particular, we do not extend Gobra's encoding into Viper as part of this goal. The purpose of the prototype is to analyze the expressiveness and limitations of our designed type system.

Evaluation. We will evaluate the expressiveness and limitations of our linear type system extension. In particular, we will evaluate in which cases the annotation overhead can be reduced by adding type annotations.

4 Extension Goals

Extend our linear type system to support closures. In order to further improve the capabilities of our linear type system in Gobra, we aim to extend our linear type system to support closures.

Application to larger Go programs. For this goal, we aim to apply our linear type system to larger Go programs, such as the VerifiedSCION [8] project. This will allow us to better evaluate the number of annotation lines and the time required for verification saved by our approach.

References

[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] 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.

[4] 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

[5] 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

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

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

[8] N. S. Group and the Information Security Group, "Verifiedscion," https://tutorials.baguasys.com/how-to-create-a-new-algorithm/.