ETH zürich

Adding Support for Generic Types in a Program Verifier for Go

Bachelor's Thesis

Colin Pfingstl

August 26, 2023

Advisors: Prof. Dr. Peter Müller, Felix Wolf

Department of Computer Science, ETH Zürich

Abstract

Gobra is an automated, modular verifier for Go programs. Gobra is based on the Viper verification infrastructure. Gobra encodes annotated Go programs into Viper programs that are then automatically verified. In this thesis, we implement the parsing and type-checking of Go generics in Gobra and evaluate our solution. Further, we improve the current implementation of type modifiers in Gobra. Type modifiers are additional type information, which Gobra attaches to types for the purpose of verification.

Contents

Contents iii

1 Introduction 1

2 Generics 3
2.1 Go Generics 3
2.1.1 Generic Types 3
2.1.2 Generic Functions 4
2.1.3 Type Constraints 4
2.2 Extending the Parser 5
2.2.1 Previous Parser Architecture 5
2.2.2 Extension 6
2.3 Extending the Type-checker 9
2.3.1 Previous Type-checker Architecture 9
2.3.2 Extension 10

3 Type Modifiers 15
3.1 Previous Architecture 15
3.1.1 Shared and Exclusive 15
3.1.2 Ghost and Actual 16
3.2 Solution 16
3.2.1 Design 17
3.2.2 Modifier Unit Interface 17
3.3 Discussion 19

4 Evaluation 21
4.1 Performance 21
4.1.1 Overhead of Generics 21
4.1.2 Scaling of Generics 22
4.2 Expressiveness 27

5 Conclusion **31**
5.1 Future Work 31

Bibliography **33**

Chapter 1

Introduction

Gobra [1] is an automated, modular verifier for Go programs. Gobra is based on the Viper verification infrastructure. Gobra encodes annotated Go programs into Viper programs that are then automatically verified. Gobra is used in the VerifiedSCION [2] and Centre for Cyber Trust [3] projects to verify correctness and security properties of substantial codebases.

With the introduction of Go 1.18, generics where added the the Go language. The main goal of this thesis is to support the parsing and type-checking of these generics in Gobra. We aim to design an efficient solution that does not significantly increase the parse and type-check time of programs without generics.

Gobra extends Go's type system by attaching additional modifiers to every type, capturing information relevant for verification. These modifiers are called type modifiers. As a second goal of this thesis, we improve the existing implementation of type modifiers in Gobra. We design a solution that makes adding new type modifiers easier. To achieve this, we unify the currently implemented type modifiers.

In Chapter 2, we discuss the abilities of generics in Go. Furthermore, we explain how we implemented the parsing and type-checking of generics in Gobra and discuss the implementation challenges we encountered. In Chapter 3, we discuss how type modifiers are currently implemented and how we improved their architecture. In Chapter 4, we discuss the performance and expressiveness of our generics implementation. In Chapter 5, we give a conclusion and give ideas for future work.

Chapter 2

Generics

2.1 Go Generics

Go supports two kinds of generics, *generic types* and *generic functions*. They allow functions and types to be parameterized by *type parameters*. The set of types with which a type parameter can be instantiated is constrained by *type constraints*. In this section, we discuss Go's generic types and functions and Go's type constraints.

2.1.1 Generic Types

Generic types are types that are parameterized with *type parameters*. An example of a generic type is shown in Fig. 2.1. In the example, `T` is a type parameter of the type `List`. In the body of the struct definition, this type parameter `T` is used like every other type. The meaning of `any` is explained later.

To use generic types in the program, all type parameters must be instantiated. See for instance Line 6 of the example. The type `List[int]` is an instance of the generic type `List`, where the type parameter `T` has been instantiated with the concrete type `int`.

```
1   type List[T any] struct {
2       next *List[T]
3       val T
4   }
5   func main() {
6       var list = List[int]{next: nil, val: 10}
7   }
```

Figure 2.1: A generic type

```
1 func equal[T comparable](a T, b T) bool {
2     return a == b
3 }
4 func main() {
5     var x = equal[int](1, 2)
6     var y = equal(1, 2)
7 }
```

Figure 2.2: A generic function

2.1.2 Generic Functions

Generic functions are functions that are parameterized with type parameters. The type parameters can then be used in the function signature and function body, just like for generic types. An example of a generic function is shown in Fig. 2.2. In the example, we define a generic function `equals`, which compares two comparable objects `a` and `b` of the same type.

Similar to generic types, the type parameters of a generic function are instantiated when the function is called. However, in contrast to generic types, the type parameters of generic functions do not always have to be provided by the programmer. Go's compiler employs heuristics to infer type parameter instantiations at compile time. Such a case is shown at Line 6 of the example. Here, Go uses the type information of the call arguments `1` and `2` to infer that the type parameter `T` is instantiated with `int`. If the heuristics fail, Go's compiler reports an error message, and the programmer must instantiate the type parameters manually. An example of an explicit type instantiation is shown at Line 5 of the example.

2.1.3 Type Constraints

As mentioned before, Go enables programmers to impose restrictions on the set of types with which a type parameter can be instantiated. More concretely, every type parameter is provided with a type constraint. Each type that can be used to instantiate the type parameter according to the type constraint is said to *satisfy* the type constraint.

Type constraints are listed after a type parameter definition. In the examples of Fig. 2.1 and Fig. 2.2, the type parameter `T` is constrained with the type constraints `any` and `comparable` respectively.

The constraint `any` is satisfied by every type. The constraint `comparable` is a special built-in type constraint. It is satisfied by types that support the equality `==` and inequality `!=` operators.

The other type constraints are: union type constraints, interface type constraints, and embedding type constraints. A union type constraint `T t1|t2|...|tn` is satisfied by a type `T` if `T` satisfies at least one of the constraints `t1` to `tn`. An interface type constraint is satisfied by types that *implement* the provided inter-

```
1    functionDecl: FUNC IDENTIFIER (signature block?);
```

Figure 2.3: Gobra's parser rules for function declarations

face. For instance, the type constraint `T interface{ f(int) int}` requires `T` to implement a method with signature `f(int) int`. The embedding type constraint is satisfied by a type that *embeds* the specified embedding type. For a detailed discussion of embedding types, we refer readers to the Go language specification [4].

2.2 Extending the Parser

To introduce Go generics to Gobra, we had to modify Gobra's parser. The overall architecture stayed the same, but some parts had to be adjusted and extended.

2.2.1 Previous Parser Architecture

Gobra uses a parser generator called ANTLR [5]. This parser generator takes as input a formal specification of a language in the form of a lexer and a grammar and outputs the Java code of the resulting parser. This generated parser builds parse abstract syntax trees (ASTs) from Gobra input programs. ANTLR also automatically generates tree walkers that programmers use to visit and transform ASTs. Gobra uses these tree walkers to translate the parse AST of a Gobra program to its preliminary AST representation. In summary, to extend Gobra's parser, we had to define three things, namely, the lexer, the grammar, and the parse AST translator.

The lexer defines how an input program is translated into *tokens*. Defining the lexer is straightforward and not discussed in this thesis.

The grammar defines how a stream of tokens is translated into a *parse AST*. The grammar is specified by a set of *rules* in ANTLR. Rules are written in an EBNF like language. Rules may contain nested rules and tokens defined in the lexer. Fig. 2.3 shows an example of a rule definition. This example shows the rule matching function declarations. In this rule, `FUNC` and `IDENTIFIER` are tokens, which are defined in the lexer. Essentially, `FUNC` corresponds to the string "func" and `IDENTIFIER` is any alphanumeric string starting with a letter. The expressions `signature` and `block` are other nested rules that define how function signatures and blocks are parsed, respectively. The `block` is modified by a `?` operator, which means that `block` must match 0 or 1 time. An example of some Go code that matches this rule is shown in Fig. 2.4.

As already mentioned, the parse tree translator translates the AST constructed by the parser from the input program into a preliminary AST representation. This AST representation is used by Gobra to iterate over the structure of the parsed

```
1    func add(x int, y int) { }
```

Figure 2.4: A simple Go function

program and to perform, for example, type-checking. The parse tree translator is also straightforward and is not further discussed.

2.2.2 Extension

To implement Go generics in Gobra, we modified the grammar and the parse tree translator. There were no changes made to the lexer. The parse tree translator changes are trivial and thus, are not discussed.

Modifying the grammar of Gobra to support Go generics was straightforward. The Go specification already defines the syntax of the Go language in a variant of EBNF. To implement Go generics, we translated most of the EBNF *production rules* of the Go specification directly to grammar rules in ANTLR. However, we had to adjust some production rules to avoid ambiguities in the parser.

Concretely, we made the following changes to the grammar:

(1) We added new rules that match definitions of type parameters and their constraints (e.g. `[T any, V int | bool]`).

(2) We modified function declaration and type definition rules such that functions and types can have type parameter definitions
(e.g. `func foo[T any]() {}`).

(3) We added and modified rules such that type parameters of generic functions and types can be instantiated (e.g. `foo[int, bool](3)` and `Bar[int]`).

In the following three subsections, the different changes are discussed in more detail.

Type Parameter Definitions

To facilitate the matching of type parameter definitions, we added the set of rules shown in Fig. 2.5. We translated these rules directly from the Go specifications without any modifications. As seen before, type parameters are declared with an identifier plus a type constraint (e.g. `T any`). Such a type parameter declaration is captured in the rule at Line 3 of the example. The rule matches a list of identifiers with the nested rule `identifierList` followed by the `typeConstraint` rule, which matches a type constraint. Notice that in Line 3, we did not use the `IDENTIFIER` token to match the name of the type parameter, but instead used the rule `identifierList` to match a whole list of identifiers. Multiple type parameters with the same type constraint may be listed together (e.g. `T, V any`). Hence, `identifierList` matches a comma separated list of `IDENTIFIER`.

```
1   typeParameters: L_BRACKET typeParamList R_BRACKET;
2   typeParamList: typeParamDecl (COMMA typeParamDecl)*;
3   typeParamDecl: identifierList typeConstraint;
```

Figure 2.5: Type parameter definition rules (simplified)

```
1   functionDecl: FUNC IDENTIFIER typeParameters? (signature block?);
```

Figure 2.6: Modified function declaration rule

Alternatively, multiple type parameters may also be declared after another (e.g. `T any, V comparable`). This is captured in the rule at Line 2 of Fig. 2.5, which matches lists of one or more parameter declarations joined with a comma. Finally, the rule at Line 1 of the example wraps the whole type parameter list with square brackets ("[" and "]"). The rules defining type constraints are omitted since the rules are straightforward.

Generic Functions and Generic Types

To be able to add type parameter definitions to functions, we changed the existing grammar rule that matches function declarations. Concretely, we added the `typeParameters` rule from Fig. 2.5, discussed in the previous section, to the `functionDecl` rule shown in Fig. 2.6. Notice that the nested `typeParameters` rule is modified with `?`, which matches the nested rule 0 or 1 time. We modified the type definition rule analogously to support generic types as well.

Type Parameter Instantiations

To be able to instantiate type parameters of functions and types, we modified the grammar rules concerning operands and expressions.

First, we translated the Go syntax specification into an ANTLR grammar specification, yielding the set of rules shown in Fig. 2.7. Notice that these rules distinguish between `index` and `typeArgs`. The rule `index` only matches one expression, which is meant to capture index expressions (e.g. array indexes). Conversely, the rule `typeArgs` matches a list of types, which is meant to capture type instantiations (e.g. `foo[int, bool]`).

The set of rules in Fig. 2.7 leads to ambiguities when parsing certain expressions. Consider the expression `base[i]`. This expression may be parsed as a type parameter instantiation of the type `base` with type argument `i`. The corresponding parse tree is the tree (a) shown in Fig. 2.8. Conversely, this expression may also be parsed as expression index (e.g. array indexing) of expression `base` with index `i`. The corresponding parse tree is the tree (b) shown in Fig. 2.8.

```
1   primaryExpr: operand | primaryExpr index | ...
2   operand: literal | operandName typeArgs? | ...
3   index: L_BRACKET expression R_BRACKET
4   typeArgs: L_BRACKET typeList R_BRACKET
```

Figure 2.7: Grammar translated from the Go specification (simplified)

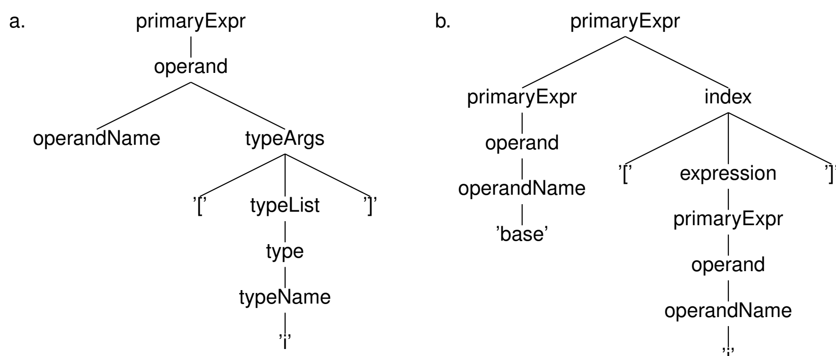

Figure 2.8: Ambiguity of the expression `base[i]`

```
1   primaryExpr: operand | primaryExpr index | ...
2   operand: literal | operandName | ...
3   index: L_BRACKET expression (COMMA expression)* R_BRACKET;
```

Figure 2.9: Modified grammar (simplified)

Since the parser has no type information available, it cannot choose between the two possible parse trees (a) and (b) in Fig. 2.8.

To eliminate this ambiguity in the parser, we modified the grammar such that it does not distinguish between type parameter instantiations and expression indexes. We deferred the disambiguation to the type-checker. Concretely, we modified the grammar as follows:

(1) We removed the extra rule `typeArgs` from the operand, meant to capture type parameter instantiations.

(2) We modified the `index` rule such that it can also capture type instantiations and not only expression indexes.

The modified grammar is shown in Fig. 2.9. With this modified grammar, there is only one possible parse tree for the expression `base[i]`, which is equal to the parse tree b. in Fig. 2.8.

2.3 Extending the Type-checker

To facilitate the type checking of Go generics in Gobra, we modified the type-checker. Again, the overall architecture of the type-checker stayed the same, but some parts were adjusted and extended.

2.3.1 Previous Type-checker Architecture

In Gobra, the type-checker has two main tasks: (1) check if a program is *well-defined* and (2) associate expressions, symbols, etc. with *types*. Specifically, Gobra's type-checker takes as input the AST of a program and outputs a type information object if the program is well-defined. The type information object maps certain *nodes* of the AST (e.g. expressions, symbols) to types. Otherwise, if the program is not well-defined, the type-checker outputs a list of errors. In the following sections, the term *typing* refers to both well-definedness checking and type mapping together.

Gobra's type-checker is split into different modules. The core modules are:

(1) the base typing module

(2) several specific typing modules (e.g. expression typing, statement typing)

(3) the ambiguity resolution module

(4) several property modules (e.g. assignability, type identity, addressability)

The *base typing module* defines how well-definedness of a program is checked. Recall that the type-checker gets as input the AST of a program. The base typing recursively iterates over the nodes of that AST to check well-definedness of the program. Concretely, a node in the AST is well-defined if all of its children nodes are well-defined and the node itself is well-defined. Depending on the type of the node (e.g. expression node, type node, statement nodes) a different *specific typing module* is used to check the well-definedness of that node.

Additionally, the specific typing modules are responsible for mapping specific types of nodes to their type (e.g. the expression typing module maps expression nodes to types).

The *ambiguity resolution module* is responsible for resolving ambiguous AST nodes. This module is discussed further in Sec. 2.3.2.

Property modules define different properties (e.g. assignability, type identity, addressability). These properties may be defined on types and expressions. Properties are used by the specific typing modules, mostly to do well-definedness checks (e.g. the assignability property is used to check whether a variable assignment is well-defined).

2.3.2 Extension

To facilitate the type-checking of Go generics in Gobra, we modified the ambiguity resolution module, some specific typing modules, and some property modules. Additionally, we introduced the notion of a type set, created a satisfiability property, and implemented type parameter inference.

Ambiguity Resolution

Recall the discussion in Sec. 2.2.2 about the two interpretations of the expression `base[i]`. The parser parses this expression as the same AST node regardless of whether it is an expression index or a type parameter instantiation. However, for the typing, the difference is relevant. For this reason, Gobra has the aforementioned ambiguity resolution module. The ambiguity resolution module resolves ambiguous AST nodes to *AST patterns*. AST patterns give additional information about the AST node. When typing a node, this ambiguity resolution is called and depending on the resulting AST pattern, the node is typed differently. For instance, ambiguity resolution is called when typing `base[i1, ..., in]`. The ambiguity resolution then determines whether the expression is a type parameter instantiation or an expression index. Depending on whether the expression is a type parameter instantiation or an expression index other well-definedness checks are made.

Concretely, we extended the ambiguity resolution to handle type parameter instantiations. Consider the expression `base[i1, ..., in]`. To resolve this, the ambiguity resolution first resolves the base node `base`. In case the AST pattern of the `base` node is a *function* or a *type name*, the expression is a type parameter instantiation. Otherwise, if n is 1 and `i1` is a type, `base[i1, ..., in]` is an expression index. We implemented this logic in the ambiguity resolution.

Satisfying Type Constraints

The introduction of type parameters requires the notion of a *type set*. Informally, the type set is the set of types a type parameter could be instantiated with. Type sets are used to check whether type constraints are satisfied and also for checking properties such as assignability. Formally, the *type set* of a type constraint is defined as the set of all types that satisfy that type constraint.

Consider the type constraint `int | bool`. According to the definition of union type constraints, this constraint is satisfied by the types `int` and `bool`. Hence, the type set of this type constraint is $\{\text{int}, \text{bool}\}$.

Consider the type constraint `interface { m() }`. According to the definition of the interface type constraint, this constraint is satisfied by all types that implement a member `m()`. Hence, the type set of this type constraint is infinitely large, because there can be infinitely many types that implement `m()`. This poses challenges on the implementation because an infinite type set cannot be constructed at runtime

To accommodate both finite and infinite type sets our implementation distinguishes between two types of type sets. *Bounded type sets* are finite type sets that contain a list of types. In contrast, the *unbounded type set* is an abstract type set that represents a type set with an infinite number of types. Note that our implementation of the type set does not exactly reflect the definition of the type set. Conceptually, an infinite type set should contain infinitely many types. But in our implementation we just treat all infinite type sets as the same abstract unbounded type set.

As a consequence, checking whether a type satisfies a type constraint is not as easy as just checking whether that type is contained in the type set of that type constraint. Consider the aforementioned type constraint `interface { m() }`. Suppose, we want to check whether a type `t1` satisfies this type constraint. It does not suffice to check only whether `t1` is in the type set of `interface { m() }` because the type set of `interface { m() }` is the unbounded type set and thus always contains `t1`. We also need to check whether `t1` implements a method `m()`. In general, a type `t1` satisfies a type constraint `c1` if:

(1) `t1` implements the member set of `c1` and

(2) `t1` is contained in the type set of `c1`

We implemented this logic in a new property called satisfiability.

Type Parameter Instantiation

Type parameter instantiations are a core mechanism of Go's generics. In this section, we discuss how type parameter instantiations exactly work and how we solved them. Note that in this section, we only consider type parameter instantiations where all type parameters are explicitly provided and are not inferred. Inferring type parameters is discussed later in Sec. 2.3.2.

Recall from Sec. 2.1, that generic functions and generic types can be *instantiated* by providing *type arguments* for the type parameters. Consider a function `func foo[T any, V Bar](x T) V {}` where `Bar` is an interface. We instantiate the generic function `foo` by providing the type argument as `foo[int, Baz]` where `Baz` is a custom type.

In order to determine if the type parameter instantiation is well-defined, the type-checker has to check two points. First, the type-checker checks that the number of type arguments provided matches the number of defined type parameters. Second, the type-checker checks for each pair of type argument and type parameter that the type argument *satisfies* the type constraint of the type parameter.

In the example above, the type-checker checks that `int` satisfies `any` and `Baz` satisfies `Bar`. In case the type parameter instantiation is well-defined, the type-checker instantiates the generic type or function with the provided type arguments. A generic type or function is instantiated by substituting type parame-

```
1   func foo[T interface{ m(); n() }](x T) {
2       var y interface{ m() } = x // valid
3   }
```

Figure 2.10: Example of case (1)

ters with the corresponding type arguments. In the example, the initial type of `foo` is `func(T) V`. Instantiating `T` with `int` and `V` with `Baz` results in the type `func(int) Baz`.

Assignability

Assignability is an important property in Go. As the name suggests, a type `t1` is *assignable* to a type `t2` if an expression of type `t1` can be assigned to a location, e.g. variable, parameter, etc., of type `t2`. This property is used in statements such as variable declarations and assignments. For instance, in a declaration `var x int = y`, it is checked that the type of `y` is assignable to the type of `x`, which is `int` in this case.

To support type parameters, we had to extend the current implementation of assignability in Gobra. Omitting some details, generics introduce three new cases, that have to be handled differently, when deciding whether a type `t1` is assignable to type `t2`:

(1) `t1` is a type parameter and `t2` is a regular interface

(2) `t2` is a type parameter

(3) `t1` is a type parameter and `t2` is not a a regular interface

First, we discuss the case (1). An example of this case is shown in Fig. 2.10. Here, we have to check that the type of `x`, which is `T`, is assignable to the type of `y`, which is `interface { m() }`. The example falls under case (1) since `y` has a regular interface type and the type of `x` is a type parameter. Recall from Sec. 2.1.3 that the interface type constraint `interface { m(); n() }` requires that `T` has to implement the interface `interface { m(); n() }`. In particular, `T` has to implement the methods `m` and `n`. To check whether or not `x` is assignable to `y`, we check that `interface{ m(); n() }` implements `interface{ m() }`. In the example, the type `interface{ m(); n() }` implements `interface{ m() }` and hence, the assignment is valid. In general, we check that `t1` implements `t2`.

Next, we discuss the case (2). An example of this case is shown in Fig. 2.11. Here we have to check that the type of `3`, which is `int`, is assignable to type `T`. The example falls under case (2) since `T` is a type parameter. Recall from Sec. 2.1.3 that the union type constraint `int | bool` requires that `T` has to satisfy `int` or `bool`. In particular, this means that `T` is either `int` or `bool`. To check whether or not `3` is assignable to `T`, we check whether `3` is assignable to *both* `int` and `bool`.

```
1    func foo[T int | bool]() {
2        var x T = 3 // invalid
3    }
```

Figure 2.11: Example of case (2)

```
1    func foo[T int | bool](x T) {
2        var y int = x // invalid
3    }
```

Figure 2.12: Example of case (3)

In the example, the type of `3` is not assignable to `bool` and hence the assignment is invalid. In general, we check that:

$$\forall s_i \in \mathsf{typeset}(c_2) : \mathsf{assignableTo}(t_1, s_i)$$

where c_2 is the type constraint of t_2, $\mathsf{typeset}(c_2)$ is the set of all types that satisfy c_2, and $\mathsf{assignableTo}(t_1, s_i)$ is a predicate that is true if and only if t_1 is assignable to s_i.

Lastly, we discuss case (3). An example of this case is shown in Fig. 2.12. Here we have to check that the type of `x`, which is `T`, is assignable to the type of `y`, which is `int`. The example falls under case (3) since `T` is a type parameter and `t2` is not a regular interface. Informally, we can say that `T` must be either `int` or `bool` according to its type constraint. To check whether or not `x` is assignable to `y`, we check whether *both* `int` and `bool` are assignable to `int`. In the example, only `int` is assignable to `int`, but `bool` is not assignable to `int` and hence the assignment is invalid. In general, we check that:

$$\forall s_i \in \mathsf{typeset}(c_1) : \mathsf{assignableTo}(s_i, t_2)$$

where c_1 is the type constraint of t_1.

Type Parameter Inference

Recall from Sec. 2.1.2 that, in certain cases, Go is able to infer type parameter instantiations of generic functions. For instance, a generic function `func foo[T any, V any](x T, y V)` can be called as `foo[int](3, true)`. In this example, only the first type parameter `T` is instantiated explicitly with type `int`. The second type parameter `V` is inferred to `bool` from the function call.

To implement this in Gobra, we modified the well-definedness check of type parameter instantiation expressions. Whenever a generic function is instantiated, the type-checker does the following: First, the type-checker checks whether the instantiation expression is a direct sub-expression of a function call (e.g.

`foo[int](3, true)`). In case it is not a direct sub-expression of a function call (e.g. `foo[int]`), no type parameters can be inferred and thus all type parameters must be explicitly provided. Otherwise, the type-checker proceeds by creating a map from type parameters to the provided type argument types and defers the well-definedness check to the outer function call well-definedness check. In the example, this map is `T -> int`. In the well-definedness check of the function call, the type-checker attempts to infer rest of the type parameters that are not already contained in the map. For this, the type-checker types all function call arguments that have a type parameter type and adds the corresponding type parameter and type pair to the map. In the example, the resulting map is `{T -> int, V -> bool}`. Finally, the type-checker syntactically substitutes every type parameter with its corresponding type mapping in the function type, which yields a type `t`, and checks that the function is fully instantiated. To check that the function is fully instantiated, the type-checker checks that all type parameters declared by `foo` are not included in `t` anymore. Type parameters that are not declared by `foo` are considered instantiated in `t`. In the example, the resulting type `t` for `foo` is `func (int, bool)`.

Chapter 3

Type Modifiers

Gobra internally uses a richer type system than Go. In particular, Gobra extends Go types with *type modifiers* that capture further information about instances of a type. In this chapter, we present a design to unify the treatment of type modifiers and discuss how we applied the design to Gobra's current type modifiers.

3.1 Previous Architecture

Currently, Gobra implements two different type modifiers, which are explained in detail in the next two sections.

3.1.1 Shared and Exclusive

In this section, we discuss the *owner modifier* of Gobra. In Gobra, an expression is either *shared* or *exclusive*. Expressions that are shared may be aliased. In contrast, expressions that are exclusive are definitely not aliased. An example is shown in Fig. 3.1. In the example, the variable `x` is exclusive. This means that `x` must not be aliased. For instance, the expression `&x` would cause an error in the type-checker. In Gobra, variables are exclusive by default. To explicitly declare a variable as shared, the `@` symbol is appended to the identifier name. In the example, the variable `y` is explicitly declared shared. Hence, the variable `y` can be aliased like in Line 4 of the example.

```
1   func foo() {
2       var x int = 16
3       var y@ int = 16

4       bar(&y)
5   }
```

Figure 3.1: Example of an exclusive and a shared variable

```
1 func foo() {
2     ghost var x int = 16
3     var y int = 4
4     y = x // not allowed
5 }
```

Figure 3.2: Example of ghost code

Currently, the logic for owner modifiers is implemented in the addressability module of Gobra's type checker. The addressability module defines whether an expression can be dereferenced.

3.1.2 Ghost and Actual

Gobra's other modifier, the `ghost modifier`, captures whether a node in the AST is part of the actual targeted program or was added solely for the purpose of verification. AST nodes that are *ghost* do not belong to the actual program. In contrast, nodes that are *actual* do belong to the actual program. This modifier prevents that the the actual program execution and its state are influenced by verification code.

An example is shown in Fig. 3.2. By default, code is classified as actual code. In the example the function `foo` and the statement at Line 3 is actual code. In contrast, `x` is explicitly marked as ghost, hence it is not part of the actual code. This implies that `x` must not influence any actual code. Line 4 of the example shows an assignment of `y` to `x`, which is not well-defined. An actual variable must not be assigned to a ghost variable, because otherwise the actual variable would be influenced by ghost code. The type-checker reports an error in this example.

To implement this separation of ghost and actual code, Gobra uses a ghost classifier and a ghost well-definedness module. The ghost classifier module implements the mapping of program nodes to the modifiers ghost or actual. The ghost well-definedness checks the well-definedness of expressions and statements in terms of modifiers. Consider the statement `y = x` from the example in Fig. 3.2. In the example, the ghost well-definedness checks that `x` is assignable to `y` in terms of modifiers, i.e. that either `y` is ghost or both `x` and `y` are actual, which is implemented in the ghost assignability module.

3.2 Solution

In Gobra's previous architecture, both type modifiers were implemented in a custom manor. In the future, further type modifiers are expected to be added to Gobra. To make adding new type modifiers easier, we investigates techniques to unify all type modifiers. A goal was to make type modifiers more maintainable

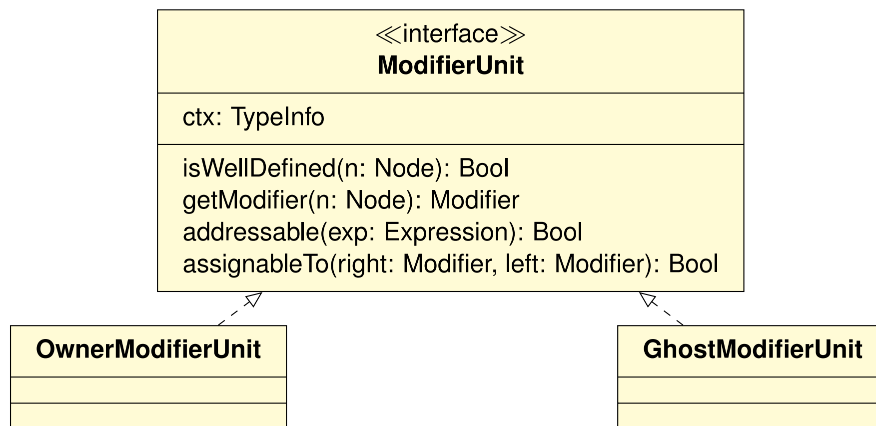

Figure 3.3: General architecture of type modifiers

and uniform. In the next two sections, we first motivate the design choices of our solution and then describe our changes to the architecture.

3.2.1 Design

As already mentioned, type modifiers are an extension of Go's type system in Gobra. In order to unify type modifiers, we first determined what all type modifiers have in common. In general, all type modifiers have their own well-definedness, modifier type mapping, and property logic. Furthermore, each modifier's typing only depends on the regular typing of Go and is independent of the typing of other type modifiers. For example, to classify whether an expression is a ghost expression is independent of whether the expression is shared or exclusive. Go's regular typing is sufficient.

With this insight, we decided to split the typing logic for each modifier in its own unit. Each modifier unit then independently implements the well-definedness, modifier type mapping and property logic of the corresponding modifier. All these units implement a common interface which we will discuss in detail in the next section.

3.2.2 Modifier Unit Interface

The `ModifierUnit` interface serves as the interface for the modifier typing logic. A simplified class diagram is shown in Fig. 3.3. The concrete modifier unit of each type modifier implements the `ModifierUnit` interface. The `ModifierUnit` interface contains two core methods, namely `isWellDefined` and `getModifier`, and methods related to properties such as `addressable` and `assignableTo`. Furthermore, each modifier unit has a reference to the normal Go typing information stored in the `ctx` field.

Well-Definedness

As already mentioned, each modifier type has its own well-definedness check. This check is implemented in the `isWellDefined` method of each modifier unit. The well-definedness check of each modifier unit assumes regular Go well-definedness. Recall from section Sec. 2.3.1, that the regular Go well-definedness is checked in the base typing module. We modified the base typing module, such that it first checks Go well-definedness and then afterwards iterates over all modifier units and uses their `isWellDefined` method to check the well-definedness of the respective modifier. Because the well-definedness check of Go is done first, modifier units are allowed to assume regular Go well-definedness.

Type Mapping

For the type mapping from AST nodes to type modifiers, we added a new modifier typing module. To get the list of modifiers for a node, the modifier typing module iterates over all modifier units and uses their `getModifier` method to get the respective type modifier.

Properties

Properties are not only defined on Go types and expressions, but can also be defined on type modifiers and expressions.

Recall the discussion about the assignability property in Sec. 2.3.2. Just like how assignability is defined on types, it is also defined on type modifiers. Hence, type modifiers pose additional requirements for expressions to be assignable to other expressions. As an example, consider assignability with ghost modifiers. As mentioned in Sec. 3.1.2, ghost state is not allowed to influence actual program state. Therefore, actual expressions must not be assigned to ghost expressions. Hence, the `assignableTo` method of `GhostModifierUnit` returns false if and only if `right` is ghost and `left` is actual. In general, to check assignability, the type-checker checks Go assignability and additionally checks assignability for each modifier unit using their `assignableTo` implementation.

Recall from Sec. 3.1.1, that addressability prevents that expressions with the owner modifier exclusive are aliased. Hence, exclusive expressions are not addressable. We moved this logic to the `addressable` implementation of the owner modifier unit. The `addressable` method of the owner modifier unit returns true if and only if the input expression has the modifier shared. Similar to assignability, to check addressability, the type-checker checks Go addressability and additionally checks addressability for each modifier unit with their respective `addressable` implementation.

3.3 Discussion

In this section, we discuss the advantages and disadvantages of the proposed solution from before.

Our solution provides a strong separation of concerns. Each type modifier is implemented in its own modifier unit. This reduces dependencies and enhances transparency. In particular, it is easier to distinguish which typing logic originates from Go and which typing logic originates from Gobra's type modifiers.

Well-definedness of type modifiers is checked more easily with the new solution. Recall that the base typing automatically checks well-definedness for each modifier unit in the new solution.

With the new solution, properties are checked more easily and code duplication is reduced. Consider as an example the assignability property. Previously, Go assignability was checked in the Go well-definedness and modifier assignability was checked in the well-definedness of the modifiers. This lead to an overlap of code in the Go well-definedness and the modifier well-definedness. Namely, code that distinguished different types of assignments. With the new solution, Go assignability and modifier assignability are both checked in the Go well-definedness, which uses the assignability interface of the modifier units to automatically check assignability for every modifier.

One disadvantage of the solution is, that finding a suitable interface for all type modifiers is not easy. Additionally, we might want to extend the interface with other properties that need to be checked in new modifiers, to reduce the aforementioned code duplication

Chapter 4

Evaluation

4.1 Performance

In the next two sections, we discuss the performance overhead of generics and how type parameter instantiations and type parameter inference scale. All tests have been conducted in WSL 2.0 on a Windows 10 machine with an i7-9700K CPU @ 3.60GHz and 32GB of DDR4-3200 RAM.

4.1.1 Overhead of Generics

To evaluate the overhead of our generics implementation, we compared the performance of the old Gobra version with the performance of our new generics implementation on the same input programs that do not contain generics. For the comparison, we evaluated the integration test suit of Gobra and the router of the VerifiedSCION project.

Fig. 4.1 shows the mean run time of 10 iterations for each project before and after generics where added. The times only include parsing and type-checking phases. The difference in run time between the old version and the generics version was 0.63 seconds for the VerifiedSCION project and 0.17 seconds for the integration test suit. The results indicate, that there is just a small overhead by our generics implementation when running programs without generics.

To investigate the run time of the integration test suit further, we consider the

Project	Old [s]	New [s]	Diff [s]
VerifiedSCION router	9.71	10.34	+0.63
Integration test suit	12.60	12.77	+0.17

Figure 4.1: Comparison of mean run times of parsing and type-checking the VerifiedSCION router and the integration test suit of 10 iterations before and after adding generics to Gobra.

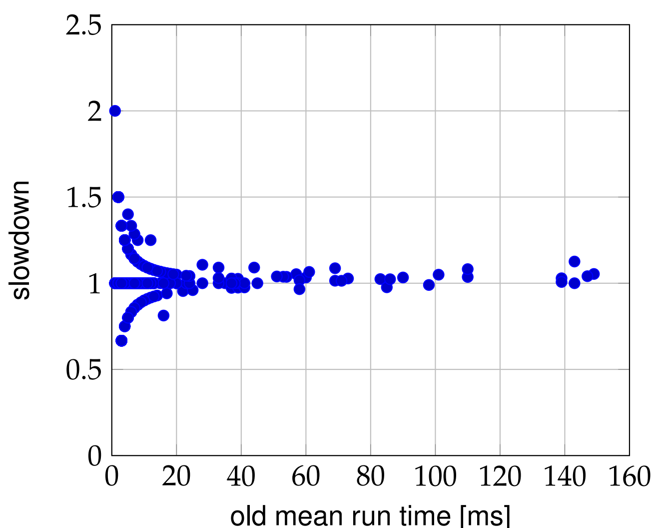

Figure 4.2: Comparison of old mean run time with slowdown of 10 iterations for each program in the integration test suit

slowdown (i.e. new mean run time divided by old mean run time) for each program in the integration test suit. Fig. 4.2 and Fig. 4.3 compare the old mean run times of the programs with their corresponding slowdown. The plot shows that the overhead on fast programs is more extreme. There are some fast programs that need twice the time with the new version. Conversely, there are fast programs that experience up to 30% run time reduction. The reduction in run time may be caused by inconsistencies in the computational load of the test machine. Slower programs with an old mean run time greater than 30 tend to have a lower overhead with the new version.

4.1.2 Scaling of Generics

To evaluate how our implementation of generics scales on type parameter instantiations and type parameter inference, we carried out a variety of different performance experiments.

Nesting Generic Structs

We investigated the influence of nesting generic structs on the run time performance. We define the *nesting factor* to be the number of recursively nested generic structs. Fig. 4.4 shows a nested struct with nesting factor 0 at Line 6 and a nested struct with nesting factor 1 at Line 7.

Fig. 4.5 shows the mean run time (parsing and type-checking) from 10 runs of nested generic structs with nesting factor 0 to 49. Increasing the nesting factor, increases the run time rapidly. Intuitively, this is what we expect. The expressions

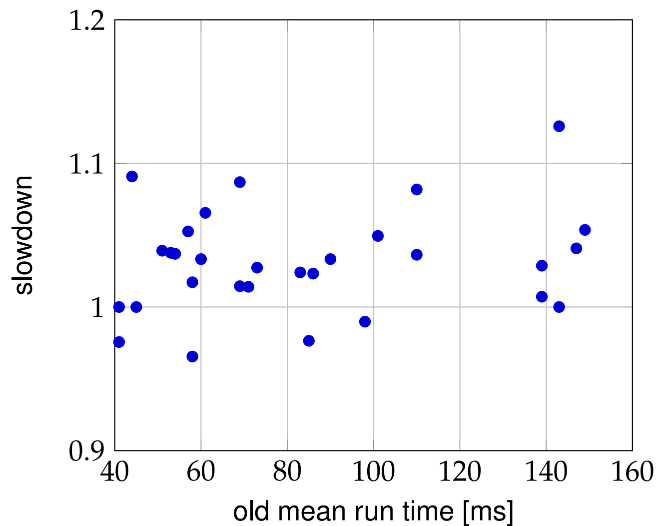

Figure 4.3: Detail view of Fig. 4.2 from old mean run time 40 ms to 160 ms

```
1    type pair[A any, B any] {
2        a A;
3        b B;
4    }
5    func main() {
6        var p0 = pair[int, int]{0, 0}
7        var p1 = pair[int, pair[int, int]]{1, pair[int, int{0, 0}]}
8    }
```

Figure 4.4: Generic struct with nesting factor 0 and 1

grow quadratically with the nesting factor, which increases the computational work of the type parameter instantiation.

Chaining Generic Function Calls

We investigated the influence of chaining generic function calls on the run time performance. We define the *chaining factor* to be the number of chained generic functions. Fig. 4.6 shows chained generic function calls with chaining factor 2.

Fig. 4.7 shows the mean run time (parsing and type-checking) from 10 runs of chained generic function calls with chaining factor 1 to 50. Increasing the chaining factor increases the run time steadily. Intuitively, this is what we expect. The number of functions and type parameters increases linearly with the chaining factor, which increases the computational work of the type parameter instantiation.

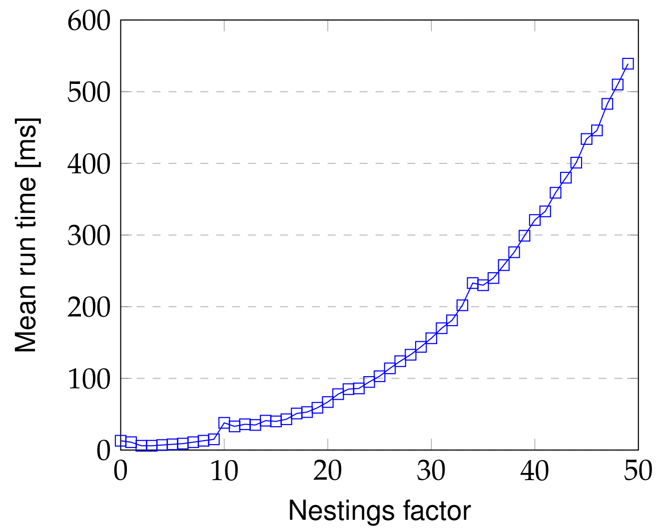

Figure 4.5: Performance of nested generic structs

```
1   func main() {
2       foo1[int](42)
3   }
4   func foo1[T1 any](x T1) { foo2[T1](x) }
5   func foo2[T2 any](x T2) { var _ = x }
```

Figure 4.6: Chained generic function calls with chaining factor 2

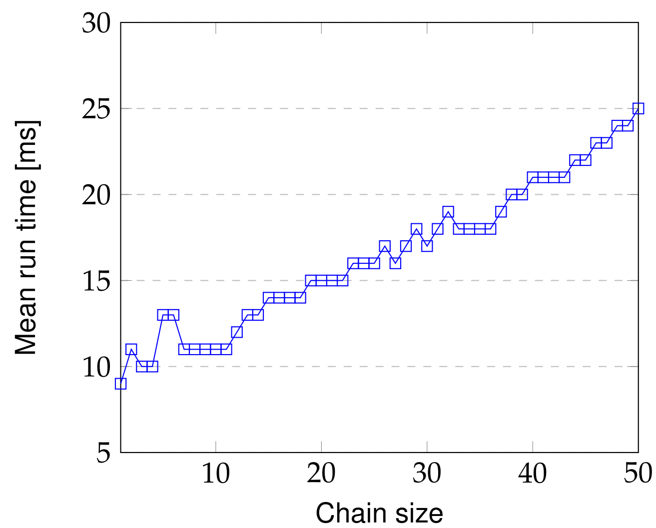

Figure 4.7: Performance of chained generic function calls

```
1   func main() {
2       foo(1, 2, 3, 4)
3   }
4   func foo[T any](x1 T, x2 T, x3 int, x4 int) {}
```

Figure 4.8: Generic function with parameter size 4 and type parameter size 2

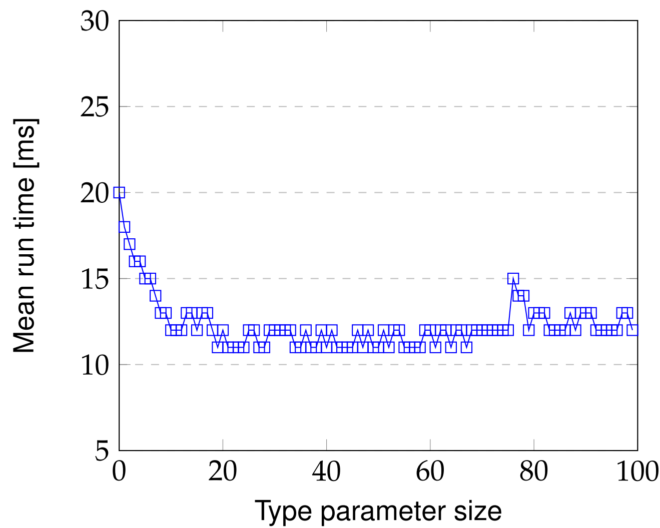

Figure 4.9: Performance of increasing type parameter size with fixed parameter size 100

Increasing Type Parameter Ratio in Function Arguments

To investigate the performance of function arguments with type parameter types, we first fix the number of parameters to a function, which we call the *parameter size*. Then we increase the amount of arguments that have a type parameter type, which we call the *type parameter size*. We restrict that all the function arguments that have a type parameter type have the same type parameter type `T`. Fig. 4.8 shows a generic function with parameter size 4 and type parameter size 2.

Fig. 4.9 shows the mean run time (parsing and type-checking) from 10 runs of generic functions with parameter size 100 and type parameter size 0 to 99. From type parameter size 0 to about 10, we see a rapid decrease of the mean run time. We suppose this is due to insufficient warm up. For parameter sizes greater than 10, the run time does not significantly increase. Intuitively, this makes sense. The amount of function arguments and type parameters is fixed. The only quantity that increases linearly with the type parameter size is the amount of function arguments that have a type parameter type. The result suggests, that increasing the amount of function arguments with a type parameter type does only lead to a small overhead.

```
1 func main() {
2     foo(1, 2, 3, 4)
3 }
4 func foo[T1 any, T2 any](x1 T1, x2 T2, x3 int, x4 int) {}
```

Figure 4.10: Generic function with parameter size 4 and type parameter size 2

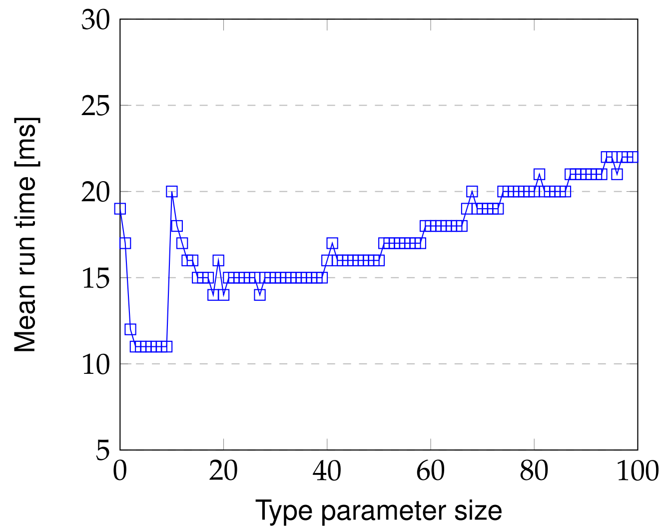

Figure 4.11: Performance of increasing inferred type parameters with fixed parameter size 100

Increasing Inferred Type Parameters

To study the performance of type parameter inference, we fix the size of function parameters to a function and increase the ratio between function parameters that have inferred type parameter types and function parameters that have concrete types. We define the *parameter size* to be the fixed number of parameters to a function and the *type parameter size* to be the number of parameters that have a type parameter type. Fig. 4.10 shows a generic function with parameter size 4 and type parameter size 2. Note that this is similar to the previous experiment, but with the difference that now the amount of type parameters of the function also linearly grows with the type parameter size.

Fig. 4.11 shows the mean run time (parsing and type-checking) from 10 runs of generic functions with parameter size 100 and type parameter size 0 to 99. Increasing the type parameter size, increases the run time steadily, except for type parameter size from 0 to around 40. We do not know what caused the irregularity in run time at the beginning. Some of the irregularity may be explained with insufficient warm up. The steady increase after type parameter size 40 is what we expect because the number of inferred type parameters increases linearly with the type parameter size.

```
1 func main() {
2     foo(4, 3, 2, 1)
3 }

4 func foo[T1 any, T2 any](x4 int, x3 int, x2 T2, x1 T1) {}
```

Figure 4.12: Generic function with parameter size 4 and type parameter size 2 (reversed)

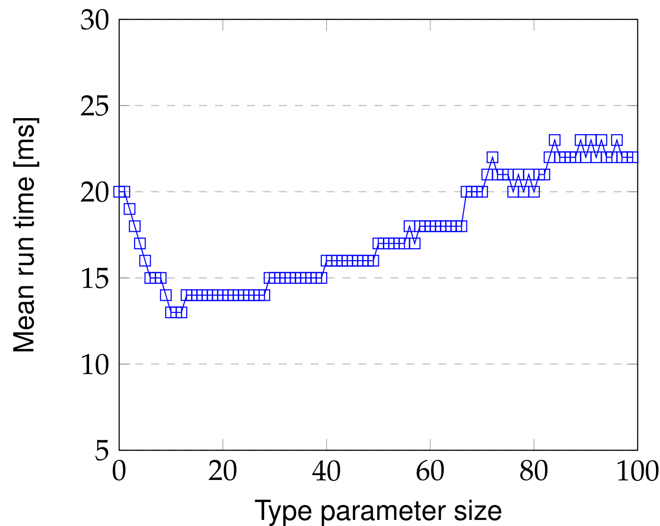

Figure 4.13: Performance of increasing inferred type parameters with fixed parameter size 100 (reversed)

Increasing Inferred Type Parameters in reverse Order

This experiment is similar to the experiment from before. The difference is, that now, type parameters are added in reverse order, beginning from the end. Fig. 4.12 shows a generic function with parameter size 4 and type parameter size 2.

Fig. 4.13 shows the mean run time (parsing and type-checking) from 10 runs of generic functions with parameter size 100 and type parameter size 0 to 99. We observe similar results to the previous experiment, except that the irregularity already ends with type parameter size greater than 10.

4.2 Expressiveness

In this section, we discuss the capabilities and limitations of our generics implementation when compared to the full generics support of Go.

In general our implementation supports generic functions and generic types. Both can be parameterized with type parameters, which have associated constraints. We implemented the union type constraint, the interface type constraint, the em-

```
1 func foo[T interface { interface { m() } }]() { }

2 type bar interface { m() }
3 func foo[T interface { bar }]() { }
```

Figure 4.14: Embedded interface type constraint inside of an interface type constraint

bedding type constraint, and the special type constraints `any` and `comparable`. Furthermore, we implemented the properties assignability, comparability, implements, type identity, type merging, and underlying type. Additionally, we created the satisfiability property. We also implemented simple type inference. In the following sections we discuss some of the limitations of our implementation.

Embedded Unnamed Interface Type Constraints

In Go, it is possible to embed interfaces as shown in Line 1 of Fig. 4.14. In the example, the interface type constraint embeds another interface type constraint that has the member `m()`. The effect of this is, that the member set of the outer interface includes the member set of the inner interface. The member set of the inner interface includes the method `m()` and hence the member set of the outer interface also includes `m()`. Essentially, this means that the resulting interface is equal to the interface `interface { m() }`. In Gobra, it is currently not possible to embed nameless interfaces. In the example, this means that Line 1 does not work correctly and must be refactored to Lines 2 and 3, such that the inner interface has a name. The limitation comes from the already existing member set calculation logic of Gobra. In practice, this problem can be easily avoided by always giving names to interfaces.

Type Constraints with Underlying Types

In Go, there is a notion of an *underlying type*. The formal definition of an underlying type is provided in the Go language specification. Informally, an underlying type is the underlying primitive type (e.g. `int`, `bool`, `[]int`, etc.) of a named type. For instance, the underlying type of a type `T1` with declaration `type T1 int` is `int` and the underlying type of a type `T2` with declaration `type T2 []int` is `[]int`.

Underlying types may be used in embedded type constraints and are denoted with "~". For example, a type constraint `interface { ~int }` is satisfied by all types that have an underlying type of `int`. In contrast, the type constraint `interface { int }` is satisfied by all types that are `int`. Our generics implementation currently does not support type constraints with underlying types.

```
1   func foo[T any](x T) {
2       bar[T](x)
3   }

4   func bar[T any](x T) { }
```

Figure 4.15: Erroneous program rejection

Limitations caused by the Type Parameter Inference

Recall from Sec. 2.3.2, that type parameters of generic function calls can sometimes be inferred. To determine, whether a function `f` is fully instantiated we check, that all type parameters declared in `f` are not included in the resulting type of the instantiation of `f`. This implementation leads to problems for some programs. Consider the example program in Fig. 4.15. This program is a valid Go program. Our implementation encounters the following problem: In the function `foo`, the type parameter `T` is used to instantiate the type parameter `T` of the function `bar`. The resulting type of `bar[T]` is `func (T)`. Our implementation then throws an error, because the resulting function type still contains a type parameter of `bar`, namely `T`, and believes the function is not fully instantiated. The problem here, is that the implementation cannot distinguish between the type parameter `T` of `foo` and the type parameter `T` of `bar`.

One approach to solve this, is to extend the type information of functions to include uninstantiated type parameters. For instance, the resulting type of `bar[T]` in the example from before could be `(func (T), [])`. Where the second element `[]` is the list of uninstantiated type parameters of that function. Further, the type of `bar` would then be `(func (T), [T])`. To check whether a function is fully instantiated, the implementation would then check that the list of uninstantiated type parameters in the resulting type of `bar[T]` is empty.

Chapter 5

Conclusion

In this thesis, we implemented parsing and type-checking a big part of Go's support for generics with a few limitations. The implementation included modifying the parser and the type-checker. We introduced new concepts including generic types, generic functions, type parameter instantiations, the notion of a type set and the satisfies property.

Further, we unified existing implementations of an extension of Go's type system. We introduced a new interface, the modifier unit, which gives a standardized access to every type modifier. Our unification provides a strong separation of concerns, makes adding new type modifiers easier, and simplifies well-definedness.

The evaluation of our generics implementation indicates that there is only a small performance overhead when parsing and type-checking programs without generics. Additionally, we studied the performance of generics and type inference of our implementation with a variety of different tests.

5.1 Future Work

Generally, there is a lot of future work for generics in Gobra.

As discussed, our implementation includes some limitations, such as underlying types and embedded unnamed interface type constraints, which are not implemented. These limitations could be addressed in future work. Additionally, the type parameter inference could be improved such that it does not impose any limitations on the programs anymore.

In this thesis, we addressed the implementation of the parser and type-checker for generics. Further, the encoding of programs containing generics to Viper programs has to be implemented to verify code containing generics.

Type parameter inference is an interesting topic for future work. The type parameter inference of Go is limited and based on simple heuristics. Gobra could extend

this type parameter inference with complex inference algorithms to make type inference more powerful. A powerful type parameter inference would make writing generic ghost code less verbose because more type parameter instantiations could be inferred and thus omitted.

Bibliography

[1] “Gobra.” [Online]. Available: https://www.pm.inf.ethz.ch/research/gobra.html

[2] “Verified SCION.” [Online]. Available: https://www.pm.inf.ethz.ch/research/verifiedscion.html

[3] “Centre for Cyber Trust.” [Online]. Available: https://www.pm.inf.ethz.ch/research/cyber-trust.html

[4] “The Go Programming Language Specification,” Dec. 2022. [Online]. Available: https://go.dev/ref/spec

[5] T. Parr, “ANTLR.” [Online]. Available: https://www.antlr.org/about.html

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

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):

Adding Support for Generic Types in a Program Verifier for Go

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

Name(s):	First name(s):
Pfingstl	Colin

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
Winterthur, 26.08.2023

Signature(s)

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