Adding Support for Generic Types in a Program Verifier for Go

Bachelor Thesis Project Description

Colin Pfingstl

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

Start: 28th February 2023
End: 28th August 2023

Introduction

Go is a general-purpose programming language. Go is strongly typed, garbage-collected, and has explicit support for concurrent programming [1].

Gobra is an automated, modular verifier for Go programs, based on the Viper verification infrastructure [2]. Gobra encodes annotated Go programs into Viper programs that are then verified. Gobra does not fully support all of Go's features. Generics are such a feature, which were introduced to Go about a year ago. The general goal of this thesis is to extend Gobra's type-checker to support Go's generics.

Context

Generics in Go

Go supports two forms of generics, *generic types* and *generic functions*. Generic types are type definitions that are parameterized with *type parameters*. An example of a generic type is shown in Figure 1. In the example, `T` is a type parameter of the type `List`. In the body of the struct definition, this type parameter `T` can then be used like every other type. The meaning of `any` is explained later. To use generic types in the program, the 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`.

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 Figure 2. Similar to generic types, the type parameter of a generic function is instantiated when the function is called. However, in contrast to generic types, the type parameters do not always have to be provided by the programmer. Go's compiler employs

```
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 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: A generic function

heuristics to infer type parameter instantiations. Such a case is shown in Line 6 of the example. If the heuristics fail, Go's compiler reports an error message, and the programmer must instantiate the type parameters manually.

It is possible to impose restrictions on instantiations of type parameters using *type constraints*. Type constraints are listed after a type parameter definition. In the examples of Figures 1 and 2, the type parameter `T` is constrained with the type constraints `any` and `comparable` respectively. The constraint `any` allows the type parameter to be instantiated with an arbitrary type. In contrast, the constraint `comparable` restricts `T` to be instantiated only with types that support the equality `==` and inequality `!=` operators. There exist other type constraints besides `any` and `comparable`. The union type constraint `T t1 | t2 | ... | tn` restricts the instantiation of a type parameter `T` to satisfy one of the type constraints of `t1` to `tn`. The interface type constraint defines that the instantiation of a type parameter must implement the specified interface. An example of an interface type constraint is `T interface{ f(int) int }` where `T` must implement a method with signature `f(int) int`. The embedding type constraint requires that the instantiation of a type parameter *embeds* the specified type. The concept of embedding types is detailed in the Go language specification [1].

Type-checking generics

The complexities of type-checkers for languages with generics differ greatly. Type-checkers that have to infer instantiations of type parameters can be rather complex. For example, the Java type system is Turing complete [3]. Conversely, type-checkers that rely on the programmer to specify all instantiations of type parameters are generally simpler. In contrast to inferring type parameters, such type-checkers only have to check that the rest of the program is consistent with the specified type parameter instantiations. This involves checking that the type parameter instantiations (1) comply with the type constraints and (2) match the arguments given to generic functions or generic type constructors.

As seen in the examples, the Go compiler generally forces programmers to specify all type parameter instantiations. As aforementioned, in limited cases, Go's compiler can infer type parameter instantiations for called functions. However, these heuristics are rather simple and fail in non-trivial cases.

Gobra has some additional language features compared to plain Go. Some of these features benefit from generics, e.g., ADTs and generic conversion functions. An example of a generic conversion function is a function which converts multi sets or sequences to sets. When extending these features with generics, we have the option to either (1) follow Go's design and hence force programmers to provide all type parameter instantiations, which increases annotation overhead, or to (2) develop a more sophisticated type-checker which tries to infer the type parameter instantiations as much as possible. In this thesis, we will explore the tradeoffs of these two options.

Viper also provides a basic form of generics. In Viper, type parameter instantiations do not have to be specified by the programmer. Viper's type-checker uses a unification algorithm to infer all instantiations of type parameters where possible. However, this unification algorithm cannot be directly applied to Gobra because it does not work in the presence of sub typing.

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

4        bar(&y)
5    }
```

Figure 3: Type attributes

Gobra type attributes

Gobra internally uses a richer type system than Go. In particular, Gobra extends Go types with *attributes* that capture further information about instances of a type. To illustrate this, consider the example shown in Figure 3. In this example, `y` might be aliased in `bar`, whereas `x` is definitely not aliased. Gobra can generate a less complex encoding if Gobra knows that a variable is not aliased. To capture this aliasing information, Gobra extends the Go type with either the attribute *shared* or *exclusive*. In this case, `x` is exclusive and `y` is shared. Values of types with the exclusive attribute are known not to alias and can therefore be encoded cheaper. Values of types with the shared attribute may alias. If we want to alias a variable, we must explicitly mark it as shared by appending `@` to the variable identifier, otherwise Gobra will throw a type error.

Currently, Gobra also has an attribute that captures whether a value is part of the actual program state or was added solely for the purpose of verification. In the future, further attributes are expected to be added to Gobra. The addition of new attributes currently requires a custom solution for each attribute to be added. We want to investigate how these custom solutions can be unified, such that the current treatment of attributes is more maintainable and such that adding new attributes is easier.

Core Goals

Implement type-checking generics in Gobra

The first core goal is to implement type-checking generics in Gobra. Gobra will be able to type-check all of Go's language features involving generics. These language features are generic types, generic functions, and different type constraints, namely `any`, `comparable`, interface type constraints, union type constraints, and embedding type constraints. Furthermore, we will extend all features additionally introduced by Gobra, for instance ADTs and conversion functions, such that they also support generics if applicable. To achieve these goals, Gobra's current parser and type-checker implementations must be extended. The implementation should be performant. In comparison to the original implementation, our implementation should not make programs that do not use generics, slower, or only by a small constant factor. In terms of architecture, we aim to create a maintainable and extendable implementation.

Implement type inference of generics in Gobra

The second core goal is to add support for inferring type parameters in Gobra. We address this goal in three different steps: (1) We investigate the design space and related work for type parameter inference, (2) we identify criteria that a type parameter inference algorithm should satisfy, and (3) we pick an existing algorithm that satisfies the criteria we defined previously and implement it. To stay compatible with Go, our solution must be able to infer exactly the same type parameter instantiations that are inferred by the Go compiler. Our implementation should show precise error messages if type parameters cannot be inferred and should be performant. In comparison to the implementation of goal 1, the implementation of this goal should not make programs slower if type parameters are not inferred, or only by a small constant factor.

Unify the treatment of attributes

The third core goal is to extend Gobra's software architecture such that it unifies the handling of type attributes. Such an architecture should make the treatment of the currently supported

type attributes well maintainable. Furthermore, the architecture should make adding new type attributes to Gobra easier.

Evaluation

The fourth core goal is to evaluate the performance and expressiveness of our implementation. We will evaluate the performance by running our implementation on large programs. Regarding expressiveness, we will analyze the limitations and capabilities of our implemented type inference.

Extension Goals

Extend the encoding to handle ADTs with generics

As mentioned before, we implement type-checking ADTs with generics in the first core goal. In this first extension goal, we extend the encoding of Gobra to handle ADTs with generics. The encoding is already known, but has to be implemented.

Extend the encoding to handle functions with generics

As a second extension goal, we extend the Viper encoding of Gobra to handle functions with generics. Part of this goal consists of finding a suitable encoding to handle functions with generics. One possible solution is to use monomorphization, where for each instantiation of type parameters the generic function is instantiated, resulting in multiple concrete functions for one generic function. To improve the verification performance, we may also want to consider more advanced solutions.

References

[1] "The go programming language specification." [Online]. Available: https://go.dev/ref/spec

[2] "Gobra." [Online]. Available: https://www.pm.inf.ethz.ch/research/gobra.html

[3] R. Grigore, "Java generics are turing complete," *SIGPLAN Not.*, vol. 52, no. 1, p. 73–85, jan 2017. [Online]. Available: https://doi.org/10.1145/3093333.3009871