Verifying concurrent Go code in Coq with Goose

Tej Chajed
MIT CSAIL

Joseph Tassarotti
Boston College

M. Frans Kaashoek
MIT CSAIL

Nickolai Zeldovich
MIT CSAIL

Abstract

This paper describes Goose, a system for writing code in Go and translating it to a model in Coq. The Coq model plugs into Iris for concurrency proofs, giving an end-to-end system for writing and verifying concurrent systems. We have used Goose as part of our work on Perennial to verify a concurrent, crash-safe mail server that gets good performance.

1 Introduction

This paper describes Goose, a subset of Go, a model of Goose in Coq, and a translator from Goose programs to the Coq model. The workflow for the developer is to write a program in Goose (which is just Go code, with some restrictions), translate it to Coq, and then finally to write a proof on top of the model using Iris [3]. We used Goose in Perennial [1] to verify a concurrent, crash-safe mail server. We've also written several other systems in Goose but which we haven't verified, including a persistent key-value store and a concurrent write-ahead log.

We developed Goose after struggling to get good performance using extraction. In our previous verified systems, we typically implemented the system in a shallow, monadic embedding in Coq, which could be extracted and run with an interpreter for the impure operations. This approach worked well, but offers little control over the generated code. When we started working on concurrency, it was difficult to get any speedup from multiple cores using extracted code and the Haskell concurrent runtime, neither of which were optimized for multicore scalability (for example, allocating more memory creates more work for the garbage collector, which doesn't have good multicore scalability).

2 Coq model

We use a shallow, monadic embedding in Coq to represent Goose programs. The Goose translator emits a Coq term of type `proc T`. This inductive type is shown in Figure 1. The `Call` constructor represents calling a primitive impure operation of Go (eg, reading and writing pointers, maps, and slices). We use monadic `Ret` and `Bind` constructors to sequence operations using arbitrary Gallina code. Finally, to support concurrency we include a `Spawn` operation to create threads. Goose places a tight syntactic restriction that every variable is assigned only once; this means that sequencing in Go can be modeled with the pure sequencing of `Bind`. This trick greatly simplifies the model, since we don't need to model creating and loading variables and only need to write the semantics for impure heap and file-system operations. Loops typically carry some state from one iteration to the next, so Goose has special support for a particular `for`-loop pattern. Whenever these limitations are too restrictive, the programmer can always allocate on the heap and load and store with the newly-created pointer.

```
Inductive proc : Type -> Type :=
| Call {T} (op: Op T): proc T
| Ret {T} (v: T): proc T
| Bind {T T'} (p1: proc T')
              (p2: T' -> proc T): proc T
| Loop {T R} (body: T -> proc (LoopOutcome T R))
             (init: T): proc R
| Spawn T (p: proc T): proc unit.
```

Figure 1. Shallow representation of Goose programs.

This representation of programs refers to primitives with the `Call` constructor, which we separately give a semantics as a transition relation. These relations happen to be written using relation combinators, but this isn't essential to the approach and any Coq relation would work.

Using a shallow embedding has several advantages: we do not have to define substitution, and during proofs, we can simplify some terms by reducing them. However, it also had some disadvantages. One feature the shallow embedding cannot express is pointers to struct fields, since structs are represented as Coq records (see §3.3). Another disadvantage is that the shallow embedding isn't natively supported by Iris (which supports arbitrary languages, as long as they use a deeply-embedded representation). To fix these issues, we have begun experimenting with a deep embedding; we've made progress in this direction which will be part of the talk.

3 Implementation

The Goose translator goes through some trouble to make sure the translation and semantics accurately captures Go. This section describes some subtleties we addressed.

CoqPL'2020, January 25, 2020, New Orleans, LA, USA
2019.

3.1 Parsing Go

The translator is implemented in Go, which means it can use the `go/ast` package to load and represent Go. Using the official support (the same packages used by many Go static analysis tools) gives us confidence that the parsing code is correct, which is important since the Go compiler and Goose translator should agree on what the source code means. We're also able to use `go/types` to get types that disambiguate what the syntax means in a couple places. Without official support it would be quite painful to parse Go, let alone type-check it; with these packages, the Goose translator is quite concise.

3.2 Using a typed model

After Goose emits some Coq code, Coq also has to type-check it. The proc T term representing a program has a type index, which enforces that terms always satisfy some basic type discipline.

Another subtlety in the Coq model is the need to sequence impure expressions. For example, the semantics of `f() + g()` depends on the order the two function calls are evaluated in. The translator does not sequence impure statements within expressions, so the programmer must do so, e.g. writing `a := f(); b := g(); return a + b`. If the program violates this sequencing discipline, then the Coq translation won't type check, since a `proc T` isn't the same as a `T`. This limitation isn't fundamental, but it avoids any reasoning about evaluation order.

3.3 Custom Datatypes

Goose supports Go struct types by translating into a Coq record type. For example, the following Go code defines a struct containing a file handle and a map:

```
type Table struct {
  Index map[uint64]uint64
  File  filesys.File
}
```

This is translated to a record in Coq, which is wrapped into a Coq module to provide namespacing:

```
Module Table.
  Record t := mk {
    Index: Map uint64;
    File: File;
  }.
End Table.
```

3.4 Modeling shared memory

Goose clearly needs a semantics for operations on pointers to be useful. However, modeling Go's shared memory support requires some care to handle weak memory: the Go memory model [2] specifies that accessing data simultaneously from multiple goroutines (lightweight threads) requires serialization, for example using locks. As long as the code follows this requirements, Go can use efficient loads and stores and yet ensure threads observe sequential consistency on top of a weak memory model like x86-TSO.

Goose enforces serialized access to shared data (pointers, slice, and maps) by making racy access to the same data undefined behavior. A *race* is formally defined as any instance of unordered accesses to the same object where at least one is a write. The Goose semantics identifies races by modeling writes, such as a store `*p = v`, as two atomic operations, a start and an end, and makes it undefined behavior for a procedure to ever overlap a write with another operation on the same pointer. When we use the Goose semantics for proofs in Perennial, we prove that the code never triggers undefined behavior, so verified code really does get a sequentially-consistent view of memory.

Reasoning about these split write operations turns out to be easy in Iris. On top of the Goose semantics we prove the usual Hoare triples for reading and writing memory. This is only possible because the capability for accessing a pointer, written $p \mapsto_n v$, represents *exclusive* access to the pointer p; threads obtain this exclusive access either by allocating a new pointer and not sharing it, or by mediating access with locks. We use a variant of the same idea to model hashmap iteration, which has a similar problem with iterator invalidation.

3.5 Unsupported features of Go

Notable features that are present in Go but not modeled in Goose include channels, interfaces, and first-class functions. These features would be difficult to model for little gain, given the low-level systems we want to use Goose for in practice.

4 Future work

We're working on porting Goose to work with a deep embedding. The approach re-uses the Iris "heap language", a simple lambda calculus with heap operations and concurrency. This model has made it easier to extend Goose to include most ordinary Go code, and natively integrates with Iris's program logic library. However, we don't have experience yet to compare proving on top of this model compared to the shallow embedding; by the time of the talk we will have more to report.

It would be great to disentagle Goose, concurrency, Iris, and crash-safety. Our use case for Goose involved crash safety, but it's not essential to make this approach useful. Similarly we are currently using Goose for concurrent systems, but it also be applied to sequential code without using Iris for the formal reasoning.

5 Conclusion

Goose is a way to verify concurrent systems written in Go, which gives low-level control and good performance. You can find the implementation, examples, and more documentation at https://github.com/tchajed/goose.

References

[1] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, Oct. 2019.

[2] Google. The Go memory model, May 2014. URL https://golang.org/ref/mem.

[3] R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems*, pages 696–723, Uppsala, Sweden, Apr. 2017.