Translating Pedagogical Exercises to Viper's Go Front-end

Bachelor Thesis Project Description

Timon Egli

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

1 Introduction

When writing code, it is important to ensure that the program is correct. Even for experienced programmers, this is very difficult to achieve, especially when working on larger projects. Testing is one of the methods used to provide some correctness guarantees. Unfortunately, testing techniques typically only cover a small subset of all possible program behaviours, and thus are not suitable to identify all errors in the program. There has been significant progress on program verification.

Program verification is a method used to prove that a program complies with a given specification. Verification is performed by a tool called a program verifier, which receives as input the program with its specification, which must be written formally and unambiguously. Gobra [1], developed at the Programming Methodology Group at ETH Zurich, is an example of such a program verifier. It allows the user to specify programs written in the programming language Go and to verify them. If Gobra succeeds, then the program is guaranteed to satisfy its specification; otherwise, an error message indicating the source of the verification error is presented at the level of the source code.

```
1   requires 0 <= n
2   ensures res == n*(n+1)/2
3   decreases n
4   func sum(n int) (res int) {
5     res = 0
6     var = i
7     invatiant i <= (n+1)
8     invatiant res == (i-1)*i/2
9     decreases n-i
10    for i := 0;i <= n;i++ {
11        res = res + i
12    }
13  }
```

Figure 1: The function "sum" written in Go computes the sum of the first n natural numbers. This function is annotated with a Gobra specification that restricts callers to only pass non-negative values to "n". The postcondition ensures that the result is n*(n+1)/2. With the "decreases" clause, Gobra is able to prove termination.

In Gobra, the specifications are usually written in the form of function contracts. In its more basic form, a function contract consists of pre and postconditions. Preconditions identify the states in which the function can be called and postconditions describe what the caller is allowed to assume after the function returns. The precondition in Figure 1 is in line 1 and the postcondition is in line 2.

Despite the fact that program verifiers are extremely effective at proving the absence of bugs, program verification is still considered an obscure and difficult field that suffers from a steep learning curve and a lack of learning resources aimed at absolute beginners.

With this project, we try to lower the entry barrier of programming verification for developers of mainstream programming languages. To that end, we plan to provide new pedagogical material aimed at Go developers. As such, we will devise introductory examples and exercises written in Go and verified in Gobra. To create more material, we will use a subset of the examples and exercises

taken from a recent book on Program Verification which uses Dafny [2], a verification-oriented programming language, and re-implement them in Go and Gobra.

2 Background

2.1 Gobra

Gobra is an automated, modular verifier for heap-manipulating, concurrent Go programs. It is a front-end of Viper and was developed at ETH Zurich. A large subset of Go is supported by Gobra, which includes subtyping via interfaces in a modular way, initialization of code via global variables, goroutines and many primitive data structures. A lot of advanced features such as concurrency primitives like channels, Mutexes and WaitGroups are also supported by Gobra. The verifier enforces properties like memory safety, crash safety, data-race freedom, as well as partial correctness based on user-provided specifications.

There are already teaching materials in Gobra like the tutorial [3], which is a quick-paced introduction to Gobra, with a big focus on its more advanced features such as interfaces and concurrency.

2.2 Viper

Viper [4] is an Intermediate Verification Language, which means that it is supposed to be a target of verifiers for other languages like Go, Rust and Python. These verifiers translate the programs written in those languages, together with their specification, into Viper and then verify it using Viper's tooling. A verifier for Rust is Prusti [5] and Nagini [6] is a verifier for Python. Viper verifies partial correctness of program statements by default, but can also verify total correctness.

The language is based on a variant of separation logic [7] and thus, allows us to reason about heap-allocated data structures. It also supports fractional permissions to be able to distinguish between read accesses and write accesses, which is useful to reason about thread interactions in concurrent programs.

```
0   method sum(n: Int) returns (res: Int)
     requires 0 <= n
     ensures  res == n * (n + 1) / 2
    {
        res := 0
5       var i: Int := 0
        while(i <= n)
            invariant i <= (n + 1)
            invariant res == (i - 1) * i / 2
        {
10          res := res + i
            i := i + 1
        }
    }
```

Figure 2: Viper method that computes the sum of the n first integer assuming that it is called in a state that satisfies n being non-negative. Termination is only guaranteed with the additional "decreases" statement.

2.3 Dafny

Dafny is a verification-aware programming language equipped with a static program verifier, originally developed at Microsoft Research. It includes constructs found in imperative programs, like loops and dynamic object allocation, but also supports features typically seen in functional programming languages, such as recursive functions and inductive data types.

Dafny has features for pre- and post-conditions, loop invariants, as well as calculational proofs. A loop invariant is a condition that is true immediately before and immediately after each iteration of a loop. A calculational proof is a proof in which the steps are justified by a series of equations. Unlike Viper and Gobra, Dafny requires termination checking. By default, Dafny rejects programs for which it cannot find a termination measure.

```
1   method sum(n: int) returns (res: int)
2    requires 0 <= n
3    ensures res == n*(n+1)/2
4   {
5     res := 0;
6     var i := 0;
7     while(i <= n)
8         invatiant i <= (n+1)
9         invatiant res == (i-1)*i/2
10    {
11        res := res+i;
12        i := i+1;
13    }
14  }
```

Figure 3: Sum of the n first integers computed in Dafny.

The programming language is used in several large verification projects like Ironclad [8] and IronFleet [9]. An advantage of Dafny's ecosystem is the availability of excellent introductory material, which is aimed at programmers with no previous exposure to program verification. Despite this, learning verification using Dafny also requires learning and getting comfortable with a new language.

3 Goals

3.1 Core Goals

As already mentioned, the aim of this project is to translate pedagogical verification exercises and examples from Dafny to Gobra. We can break this down into two sub-goals.

3.1.1 Implement a subset of examples and exercises from the book Program Proofs in Gobra

Program Proofs is an introductory book about Dafny aimed at complete beginners in program verification. It was written by K. Rustan M. Leino and has not yet been published. Given its target audience, the book presents verification from the most basic concepts and develops them at a slow pace in multiple examples and exercises. Our goal is to identify and implement a subset of the examples and exercises from the book in Gobra. Initially, our focus will be on lists (Chapter 6), unary numbers (Chapter 7), loops (Chapter 10), data-structure invariants (Chapter 11), recursive specifications (Chapter 12), and arrays and searching (Chapter 13). Given that the results of this thesis are meant to be used as teaching material, our implementations will be carefully documented. In particular, we will point out all cases where the Gobra implementation differs from the one in Dafny, for example, due to missing features in Gobra and, whenever adequate, we will do multiple translations of the Dafny code into Gobra, including a literal translation and more idiomatic translations that make use of common idioms in Go and Gobra.

3.1.2 Code comparison and identification of missing features

Another outcome of this thesis is to devise a comprehensive comparison between Dafny and Gobra based on the examples and exercises that we target. In particular, the comparison will focus on the programming capabilities of the underlying language, as well as the expressiveness of the specification and verification features. In the course of this project, will find out what features Dafny has that Gobra does not and thus makes the Gobra proof cumbersome or even impossible. Finally, we will also take note of the features provided by the respective IDEs of both Dafny and Gobra with the goal of identifying useful features that could be implemented in the Gobra IDE [10].

3.2 Extension Goals

If there is still some time left after the core goals, there are interesting extension goals to consider.

3.2.1 Verifying additional chapters

Since the book contains 18 chapters in total, another suitable extension goal could be to translate the remaining parts of the book into Gobra.

3.2.2 Feature implementation

Having compared the features available in both Dafny and Gobra, we plan to select a subset of the features identified in sub-goal two, and implement them in Gobra. Tentatively, we expect features like *calc-blocks*, *assert-by statements* and *(non-deterministic) match statements* to be candidates for being implemented in Gobra. The extension goal includes testing the implemented features extensively to ensure that their implementation is reliable.

3.2.3 Feature evaluation

We take a look at the features implemented in the previous extension goal and evaluate them. This includes writing a text in which we reflect on the difficulties we encountered and also whether we are satisfied with the result.

3.2.4 Web page with interactive tutorial

An idea would be to build a website where users can read the solutions of the book for multiple Viper front-ends side by side, in the style of the web-book Counterexamples in Type Systems [11] and even run it directly from the webpage. This way, beginners to program verification could try out the tools directly from their browsers, without installing any additional software.

References

[1] Gobra: Modular Specification and Verification of Go Programs. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17

[2] The Dafny Programming and Verification Language. [Online]. Available: https://dafny.org/

[3] Gobra Tutorial. [Online]. Available: https://github.com/viperproject/gobra/blob/master/docs/tutorial.md

[4] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[5] Prusti. [Online]. Available: https://www.pm.inf.ethz.ch/research/prusti.html

[6] Nagini. [Online]. Available: https://github.com/marcoeilers/nagini

[7] Separation Logic: A Logic for Shared Mutable Data Structures. [Online]. Available: https://www.cs.cmu.edu/~jcr/seplogic.pdf

[8] Ironclad Apps: End-to-End Security via Automated Full-System Verification. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

[9] IronFleet: Proving Practical Distributed Systems Correct. [Online]. Available: https://www.andrew.cmu.edu/user/bparno/papers/ironfleet.pdf

[10] Gobra IDE. [Online]. Available: https://github.com/viperproject/gobra-ide

[11] Counterexamples in Type Systems Logic: Unstable type expressions. [Online]. Available: https://counterexamples.org/unstable-types.html