ETH zürich

Translating Pedagogical Exercises to Viper's Go Front-end

Bachelor's Thesis

Timon Egli

April 09, 2023

Advisors: João Pereira, Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

Gobra is a program verifier for the Go programming language. 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. Apart from a tutorial that focuses on the more advanced features of Gobra, there is a lack of teaching material, especially for beginners.

With this thesis, we create new pedagogical material aimed at Go developers without prior exposure to verification. Our approach involves translating a subset of examples and exercises from the book `Program Proofs`, an introductory book on program verification in Dafny, to Gobra. Furthermore, we document the code to explain the features used and how they differ from Dafny. We also devise a comprehensive comparison between Dafny and Gobra based on the translated examples and exercises.

Contents

Contents iii

1 Introduction 1

2 Background 3
2.1 Dafny . . . 3
2.2 Gobra . . . 8

3 Translating the Examples to Gobra 13
3.1 Chapter 6: Lists . . . 13
3.2 Chapter 7: Unary Numbers . . . 14
3.3 Chapter 8: Sorting . . . 17
3.4 Chapter 10: Data-Structure Invariants . . . 17
3.5 Chapters 11 & 12: Loops and Recursive Specifications, Iterative Programs . . . 19
3.6 Chapters 13 & 14: Arrays and Searching and Modifying Arrays . . . 19

4 Evaluation 23
4.1 Gobra vs Dafny . . . 23
4.1.1 Inconsistency in the Ghost Code . . . 23
4.1.2 Correspondence Between Functions, Methods and Lemmas 23
4.1.3 Differences in Annotation Overhead . . . 24
4.1.4 Missing or Incomplete Features . . . 26
4.2 Gobra IDE vs Dafny IDE . . . 27

5 Conclusion 29

Bibliography 31

Chapter 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 behaviors, and thus are not suitable to prove that a program is correct. In contrast to testing, program verification can be used to ensure that all behaviours of a program comply with a formal specification. Verification is performed by a tool called a program verifier, which receives as input the program and its specification, which must be written formally and unambiguously, and outputs whether the program satisfies the specification. Gobra [3], 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      i := 0
7      invariant i <= (n+1)
8      invariant res == (i-1)*i/2
9      decreases n-i
10     for ; i <= n; i++ {
11         res = res + i
12     }
13     return res
14 }
```

Figure 1.1: 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 passing non-negative values to `n`. The postcondition ensures that the result is `n*(n+1)/2`.

In Figure 1.1 we exemplify how to specify the function `sum` that computes the sum of the first `n` natural numbers. To that end, we introduce preconditions, with the `requires` keyword (line 1) which imply that this function can only be called with non-negative values and we provide postconditions with the keyword `ensures` in line 2, which establishes an obligation for the implementer of this function that certain properties hold true when the function returns.

Finally, we specify that this function must terminate when it is called. To that end, we introduce a `decreases` clause (line 3) which specifies a termination measure.

As typically seen in these kinds of tools, Gobra expects that the user provides loop invariants to over-approximate the behaviour of the loop. To prove that the loop terminates, we annotate the loop with a decreases clause that contains a variant, i.e., an expression that decreases in every iteration of the loop.

Even though program verifiers are extremely effective at proving the absence of bugs, program verification is still considered an obscure and difficult field with 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 program verification for developers of mainstream programming languages. To that end, we plan to provide new pedagogical material aimed at Go developers without prior exposure to verification. As such, we will devise introductory examples and exercises written in Go and verified in Gobra. To that end, we will analyse a subset of the examples and exercises taken from an early draft of the book *Program Proofs* [10] on Program Verification which uses Dafny [9] [2], a verification-oriented programming language, and re-implement them in Go and Gobra. It is an introductory book on program verification aimed at developers without any experience in program verification. 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. The version of Dafny that is used in the book is prior to version 4.0 [1], and thus the examples may differ from what the code would look like in the most recent version.

The book is split up into three parts, with the first part teaching the basic concepts like assert statements, method contracts, ghost code and inductive data types. Additionally, this part introduces multiple features and idioms that help with structuring proofs: it distinguishes intrinsic and extrinsic styles for specification, and presents lemmas and calc-blocks. The second part of the book introduces features to write purely functional programs, which are useful for specifying the behaviour of programs, including those written in an imperative style. These techniques are exemplified in several examples, that range from simple list modifications to the implementation of complex data structures with invariants. The last part of the book focuses on specifying and verifying imperative programs. We are shown how to deal with programs with loops and programs that mutate the heap. An example of such an algorithm would be sorting an array.

Chapter 2

Background

The focus of this chapter is to provide the technical background for the thesis. Section 2.1 contains an overview of important features of Dafny used throughout this project. Following this, we introduce the Gobra verifier and some of its most important features.

2.1 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 was used in several large verification projects like Ironclad [7] and IronFleet [6]. 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.

Dafny's Features

The following section is an introduction to some of Dafny's most important features. We will refer back to it when talking about the translation in chapter 3 of the report.

Preconditions, Postconditions and Loop Invariants

Preconditions are defined using the `requires` clause and they are used to restrict the input space of the method. Post conditions are annotated with the `ensures` clause and they specify the expected output of the method. A loop invariant is a condition that is true immediately before and immediately after each iteration of a loop.

```
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         invariant i <= (n+1)
9         invariant res == (i-1)*i/2
10    {
11        res := res+i;
12        i := i+1;
13    }
14  }
```

Figure 2.1: Sum of the `n` first integers computed in Dafny.

Termination

By default, Dafny tries to prove that the program terminates and it fails if it is not able to prove it. To that end, it uses heuristics to infer the termination measure automatically. Unlike the example in Gobra in Figure 1.1 , the Dafny version does not require a `decreases` clause. Nonetheless, such a clause can be explicitly provided in cases where Dafny's heuristics are insufficient to infer a termination measure.

Algebraic Data Types

ADTs are types that are defined inductively. Inductive proofs of lemmas can take advantage of this property by using case distinction and applying the lemma on the smaller part recursively. Dafny's ADTs are parameterizable, which means that they allow generic data types. ADTs can be defined using the `datatype` keyword. Figure 2.2 shows an example of an algebraic data type called `List` that can represent a sequence of elements of some generic type `T`. It can either be an empty list `Nil` , or a non-empty list `Cons` that consists of a `head` element and a `tail`. The `tail` is a `List` data type as well and represents the rest of the sequence.

```
1  datatype List <T> = Nil | Cons(head: T, tail: List<T>)
```

Figure 2.2: A list represented as an ADT.

Lemmas

Figure 2.3 defines an algebraic data type called `List`. As mentioned in Section 2.1 , it can either be an empty list `Nil` , or a non-empty list `Cons`. Another function that is defined is the `Append` function that takes two `List` arguments and returns a `List`. The returned list represents the concatenation of the two arguments. The definition from line 7 to 9 is a lemma called `AppendNil` that states that appending `Nil` to any list `xs` is the same as `xs`.

```
datatype List <T> = Nil | Cons(head: T, tail: List<T>)

function method Append<T>(xs: List<T>, ys: List<T>): List<T>
    ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
    match xs
    case Nil => ys
    case Cons(x, tail) => Cons(x, Append(tail, ys))
}

lemma AppendNil<T>(xs: List<T>)
  ensures Append(xs, Nil) == xs
{}
```

Figure 2.3: Lemma stating that a list appended by an empty list is a list.

Lemmas are declared using the `lemma` keyword, and the `ensures` keyword is used to specify the postcondition that the lemma needs to prove.

Automatic Induction

Dafny is often capable of automatic proofs by induction in a totally automatic way. In some cases, the programmer still needs to help the tool by providing additional facts and instantiating theorems/lemmas explicitly to complete the proof. The lemma in Figure 2.3 is proven by Dafny's automatic induction feature and, therefore, the curly braces can be left empty. Figure 2.4 exemplifies how the proof would be done with the automatic induction tool turned off.

```
datatype List <T> = Nil | Cons(head: T, tail: List<T>)

function method Append<T>(xs: List<T>, ys: List<T>): List<T>
    ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
    match xs
    case Nil => ys
    case Cons(x, tail) => Cons(x, Append(tail, ys))
}

lemma{:induction false} AppendNil<T>(xs: List<T>)
    ensures Append(xs, Nil) == xs
{
    match xs
    case Nil =>
    case Cons(x, tail) =>
        calc {
            Append(Cons(x, tail), Nil);
            ==
            Cons(x, Append(tail, Nil));
            == {AppendNil(tail);}
            Cons(x, tail);
            ==
            xs;
        }
}
```

Figure 2.4: Manual induction proof of the `AppendNil` lemma.

Calc Blocks

Dafny provides the possibility of writing proofs in a calculational style, where the proof consists of a sequence of terms related by an operation, such as an equality or an implication. The reasoning behind each of the steps can also be integrated into the proof. Figure 2.4 shows the calc block feature (lines 15-23) that helps to make the proof more readable. A calc block can be defined using the `calc` keyword.

Function, Method and Function Method

Dafny offers many features that resemble an imperative programming language. One of them is the method, which is a piece of imperative, executable code. A method has a body that consists of a series of statements, and it can have multiple parameters and return types. However, it does not necessarily have to return something. Methods can receive mutable data structures as arguments and they are allowed to read from and write to them. In other words, they are allowed to modify a part of memory space. A method is defined using the `method` keyword.

Methods are very different from functions, because functions in Dafny define mathematical functions. As a consequence, they can not have any side effects to ensure that every time a function is called with the same input, the same output is returned. Dafny's functions can have multiple parameters and return types. By default, functions are ghost and can therefore only be used in specifications. They are defined using the `function` keyword.

Because functions are ghost by default, and can therefore not be compiled, Dafny provides the concept of function methods. They are the non-ghost counterpart to functions and have the advantage of being compilable.

Ghost Code

Ghost code is a feature of Dafny that helps programmers to write correct code. Using ghost code, we can specify additional constraints that help Dafny to prove the correctness of the program. Ghost code is only used for verification purposes and cannot be compiled, which means that it does not add any additional overhead at run time. The `AppendNil` lemma in Figure 2.4 is an example of ghost code. Other examples of ghost code include ghost functions and ghost variables which are declared like normal declarations, annotated with the `ghost` keyword.

Module System

The purpose of a module is to encapsulate methods, types, lemmas and other definitions. They are used to create reusable components that can be accessed from other programs. Each module contains an export set, which restricts the visibility of the definitions to only those that are added to the set. By adding a

`provides` clause to the name of the definition in the export set, as exemplified in 13 of Figure 2.5 , only the signature is accessible by the client. On the other hand, if the `reveals` clause is added instead, both the signature and the body of the definition become visible to the client.

```
1  module PriorityQueue {
2  ...
3  function method Empty(): PQueue  {
4      Leaf
5  }
6  predicate method IsEmpty(pq: PQueue) {
7      pq == Leaf
8  }
9  predicate IsMin(y: int, s: multiset<int>) {
10     y in s && forall x :: x in s ==> y <= x
11 }
12 export
13     provides IsEmpty
14     reveals IsMin
15 }
```

Figure 2.5: Module containing one function method and two predicates.

Dynamic Frames

Typically, functional contracts describe how some locations in the memory are changed by the call to the method, but rarely do they describe which values have remained the same. This is problematic when not all memory locations in use are known to the developer, and thus, the functional contract cannot enumerate all of the memory locations whose value was unaffected. This problem is often called the frame problem, and it is a crucial problem in program verification. To deal with this, Dafny employs the theory of Dynamic Frames [8].

```
1  method SetToC(a: array<int>, c: int)
2      modifies a
3  {
4      var i := 0;
5      while i < a.Length
6          invariant 0 <= i <= a.Length
7          invariant forall j :: 0 <= j < i ==> a[j] == c
8      {
9          a[i] := c;
10         i := i + 1;
11     }
12 }
```

Figure 2.6: Dafny method that sets the values of the array to a certain value.

A method that reads from a mutable data structure, passed as an argument, needs

to be annotated with a `reads` clause to indicate which parts of the memory are accessed by the method. This way, Dafny is able to set up a read frame. When a method writes to a specific heap location, the `modifies` clause has to be used instead to specify which memory locations are modified. Dafny is able to set up a write frame with this information. Figure 2.6 shows an example of a method that writes to a mutable data structure. The annotation required is the `modifies` clause in line 2.

2.2 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, global variables and static initializers, closures, goroutines and many primitive data structures. Advanced concurrency primitives like channels, Mutexes and WaitGroups are also supported by Gobra. The verifier enforces properties like memory safety, absence of panics, data-race freedom, as well as functional correctness.

There are already teaching materials in Gobra like the tutorial [4] , which is a quick-paced introduction to Gobra, with a big focus on its more advanced features such as interfaces and concurrency.

As shown in Figure 1.1 , Gobra's annotations are very similar to those of Dafny. The following subsections introduce some of the features that are supported by Gobra.

Functions and Methods

Go supports the concept of a function as well as the concept of a method. The difference between the two is very subtle, in that a method has a receiver type and a function does not. The receiver type is usually an object that can be used and modified by a method. Functions and methods are both defined using the `func` keyword.

Pure Functions and Methods

A function or a method in Gobra can be defined as pure if it is deterministic and does not have side effects. Writing to memory space is therefore forbidden. An advantage of pure definitions is that they can be used in specifications. In contrast to pure definitions, non-pure ones are allowed to modify the memory space.

Defining functions and methods as pure brings some restrictions with it. One of them is that they must have the from `return e` , where `e` is a single expression. Another one is that they must have exactly one return type, and that pure functions

or methods are only allowed to call other pure definitions. A function or a method can be defined as pure using the `pure` keyword.

Ghost Code

Gobra's notion of ghost code is similar to the one in Dafny, in that it is also only used for verification purposes and cannot be compiled. If a whole function or method is defined as ghost, it means that all of the in- and output parameters are implicitly ghost. It also means that these functions and methods are allowed to use constructs that are available in Gobra but not in Go, such as ADTs or sequences. Ghost code can be defined using the `ghost` keyword.

Packages

Packages are a way to encapsulate definitions and reuse their functionality in other programs, much like the module feature that Dafny provides. Go has two levels of accessibility, public and private. Public functions or methods are indicated by a name that starts with an uppercase letter, and they can be accessed from outside the package. Unlike Dafny, the programmer does not have fine-grained control over the symbols that he imports. Importing a package immediately brings all public symbols such as methods, constants and variables into context. Private functions or methods start with a lowercase letter.

Algebraic Data Types

In the course of our bachelor thesis, the implementation of ADTs was merged into Gobra. It was developed by another student as part of his own bachelor thesis [5]. Unlike Dafny, this implementation does not support polymorphism yet. Due to the fact that Go does not provide a feature for ADTs, they can only be used in ghost code. Gobra supports defining ADTs using the `type` keyword to define new types in Go, as exemplified in Figure 2.7.

```
1  type List adt {
2      Cons {
3          head int
4          tail List
5      }
6      Nil {}
7  }
```

Figure 2.7: A list represented as an ADT in Gobra.

Domain

Domains can be used to axiomatize mathematical data-structures in Gobra. They can contain multiple functions and domain axioms. Figure 2.8 exemplifies how a domain can be used to define a pair of boolean types.

```
1  type boolPair domain {
2      func left(boolPair) bool
3      func right(boolPair) bool
4      func pair(bool, bool) boolPair

5      axiom {
6        forall p boolPair :: {left(p)} {right(p)} p ==
7          pair(left(p),right(p))
8      }

9      axiom {
10       forall l, r bool :: {pair(l,r)} l ==
11         left(pair(l,r)) && r == right(pair(l,r))
12     }
13 }
```

Figure 2.8: Domain used to define a boolean pair.

Let Binding

Having an expression that enables variable binding within its own scope can be useful. The feature of Gobra that provides this functionality is the let binding. It is shown in Figure 2.9 , where `x` stores the value of the expression `expr1` and can be used in `expr2`.

```
1  let x := expr1 in expr2
```

Figure 2.9: The Let binding in Gobra.

Implicit Dynamic Frames

Unlike Dafny, Gobra is based on the theory of Implicit Dynamic Frames [11] , which proposes a different way of specifying heap-manipulating code based on permissions to memory locations. Each heap location is associated with an access permission that can be held by a method execution. A method that currently has permission to access certain heap locations is allowed to transfer them to another method through a method call. The access permission is transferred back to the caller when the called method returns.

Gobra differentiates between read and write permissions. Only a single method can own a write permission to a memory location, and no other method is allowed to own a read permission to it at the same time. A write permission to a memory location `x` can be specified with the accessibility predicate `acc(x, 1)` , or in short-

hand `acc(x)`. In contrast, reading from memory does not require full permission. A read permission to `x` is indicated with the accessibility predicate `acc(x, f)` , where `f` is a fractional value between 0 and 1.

Figure 2.10 exemplifies how the accessibility predicate can be used to get write permissions to all memory addresses of the elements of a slice. Line 1 specifies as a precondition that, in order to call the function, full access permission to all memory locations of the slice needs to be provided. In line 2, a postcondition makes the promise to the caller that all permissions will be returned. If the function contains a loop, the invariant in line 5 is required as well.

```
1  requires forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
2  ensures forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
3  func SetToC(a []int, c int) {

4      invariant 0 <= j && j <= len(a)
5      invariant forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
6      invariant forall i int :: 0 <= i && i < j ==> a[i] == c
7      for j := 0; j < len(a); j += 1 {
8          a[j] = c
9      }
10 }
```

Figure 2.10: Gobra function that sets all elements of the slice to a certain number.

Chapter 3

Translating the Examples to Gobra

Whenever we translated the examples and exercises from Dafny to Gobra, we made a clear distinction between a direct translation and a more idiomatic one. We wanted to include idiomatic translations because they should be easily understandable by programmers that are already familiar with Go.

In the upcoming sections we will talk about the main challenges in the translation of each chapter as well as the various solutions that we implemented.

Dafny	Gobra
$0 \leq i \leq 100$	$0 \leq i$ `&&` $i \leq 100$
`x.Name?` to check if the `x` is an instance of the Name variant	`x.isName` to check if the `x` is an instance of the Name variant
`if-else` expression	`cond ? e1 : e2` operator

Table 3.1: Some of the syntactical differences between Dafny and Gobra.

3.1 Chapter 6: Lists

In this chapter, one of our challenges was to translate Dafny's ADTs to Gobra. Initially, we tried to create an ADT using the domain feature of Gobra because the support for ADTs in Gobra had not been merged yet. We implemented constructors, destructors and different variants. This worked quite well, but we had to assume termination because there was no initial rank order defined on the domain. In other words, Gobra could not infer that the part that we get by applying the destructor on an ADT instance is strictly smaller than the instance itself, and therefore could not prove termination. A bachelor thesis addressed this issue by implementing a rank order on domains [5]. However, this made the code more complicated, so we chose to simplify things by assuming termination.

Once the support for ADTs was made available in Gobra, we switched to the more

idiomatic translation, where ADTs in Dafny were translated directly to ADTs in Gobra. However, at the time of writing, Gobra still does not support parametric ADTs. As such, all examples of ADTs with type parameters in the book have been translated to a monomorphized version in Gobra, where all type parameters are instantiated with `int`.

Additionally, Gobra was not able to prove termination of functions and methods that used ADTs. We realized that this problem was similar to the rank order problem on domains, and were able to make changes in the ADT implementation that allows the built-in `len` function of Go to map an ADT instance onto a number. We enforced that the `len` function must always map an ADT instance to a value that is strictly greater than the value of any of its sub-ADTs. This allowed us to use the `len` of the decreasing ADT instance as a termination measure.

Finally, we made one more compromise in the accuracy of the translation. As already mentioned, ADTs are only available in Gobra but not in Go. Therefore, we had to define all of the pure functions that used ADTs as ghost, making them non-compilable. In Dafny, however, ADTs can be used for defining new types that can be used for computing and for specification purposes.

3.2 Chapter 7: Unary Numbers

The main challenge of this chapter was to find a correct translation for Dafny's lemma. Initially, we thought that a lemma should be translated to a pure ghost function. However, we soon realized that this approach was too restrictive, as some lemmas might modify our view on predicates, making them non-pure in Gobra. Therefore, we decided to translate lemmas to ghost functions instead.

Unfortunately, this led to problems when we encountered examples of Dafny functions in the book that called a lemma. The translation to Gobra resulted in an error because a pure ghost function, which is the equivalent to a function in Dafny, is not allowed to call a ghost function in Gobra. This is because ghost functions are not pure, and Gobra only allows calls from pure functions to pure functions. We concluded that our initial approach of translating lemmas to a pure ghost function was better. We adapted our code to use this translation. For the lemmas that modified our view on predicates, we made a compromise and decided use ghost functions.

As mentioned in Section 2.2, defining pure functions or methods comes with certain syntactical restrictions. Although Gobra only allows one return type per pure function, we were able to return multiple ones by returning a struct with multiple fields. These fields contained the values that we wanted to return. Of course, the return type of the example needed to be adjusted accordingly.

In order to call a lemma and return an expression within the same pure function, we used an additional technique. As explained in Section 2.2, pure functions in

```
1  datatype Unary = Zero | Suc(pred: Unary)

2  predicate Less(x: Unary, y: Unary) {
3      y != Zero && (x.Suc? ==> Less(x.pred, y.pred))
4  }

5  function Sub(x: Unary, y: Unary): Unary
6      requires !Less(x, y)
7  {
8      match y
9      case Zero => x
10     case Suc(y) => Sub(x.pred, y)
11 }

12 lemma SubStructurallySmaller(x: Unary, y: Unary)
13     requires !Less(x, y) && y != Zero
14     ensures Sub(x, y) < x
15 {

16 }

17 function DivMod(x: Unary, y: Unary): (Unary, Unary)
18     requires y != Zero
19 {
20     if Less(x, y) then
21         (Zero, x)
22     else
23         SubStructurallySmaller(x, y);
24         var (d, m) := DivMod(Sub(x, y), y);
25         (Suc(d), m)
26 }
```

Figure 3.1: Dafny function that calls a lemma and returns an expression.

Gobra are restricted to being of the form `return e`, where `e` is a single expression. Since the lemma and the expression we wanted to return counted as two, we used Gobra's let binding to bring the context of the lemma call into the expression we wanted to return, turning it into a single expression.

In this chapter, we also encountered Dafny's predicate for the first time. It seemed reasonable to assume that the translation from Dafny's predicate to Gobra is the predicate that Gobra provides. In fact, they are two totally different concepts. The predicate in Dafny is defined as a pure function that returns a boolean value, whereas Gobra's predicate corresponds to a predicate in separation logic that can abstract over resources like heap locations. Therefore, we decided to translate Dafny's predicate to a pure ghost function.

Figure 3.1 shows an example of a function in Dafny that calls a lemma before returning. The `Unary` ADT can either be `Zero`, which represents the number 0, or it can be a `Suc`, which represents the successor of the previous `Unary`. The `Less` predicate returns true if the value of the first `Unary`, passed as an argument, is closer to `Zero` than the value of the second one. The `Sub` function subtracts two `Unary` values from each other, if the minuend is not smaller than the subtrahend. `SubStructurallySmaller` is a lemma that requires the `Unary` value `x` to be greater than the `Unary` value `y` and that `y` is not `Zero`. If this is given, the lemma ensures that `x` minus `y` is smaller than `x` itself. `DivMod` is the function that calls a the

```
1  type Unary adt {
2      Zero {}
3      Suc {
4          pre Unary
5      }
6  }
7  ghost
8  requires !x.Less(y)
9  ensures y == Zero{} ==> res == x
10 ensures y != Zero{} ==> len(res) < len(x)
11 decreases len(x)
12 pure func (x Unary) Sub(y Unary) (res Unary) {
13     return match y {
14         case Zero{}:
15             x
16         case Suc{?pre}:
17             x.pre.Sub(pre)
18     }
19 }
20 ghost
21 decreases len(x)
22 pure func (x Unary) Less(y Unary) bool {
23         return y != Zero{} && (x.isSuc ==> x.pre.Less(y.pre))
24 }
25 ghost
26 requires !x.Less(y) && y != Zero{}
27 ensures x.Sub(y).Less(x)
28 ensures res
29 decreases len(y)
30 pure func (x Unary) SubStructurallySmaller(y Unary) (res bool) {
31         return x.SubCorrect(y) && x.Sub(y).LessCorrect(x)
32 }
33 type PairOfUnary struct {
34     a, b Unary
35 }
36 ghost
37 requires y != Zero{}
38 decreases len(x)
39 pure func (x Unary) DivMod(y Unary) PairOfUnary {
40         return x.Less(y) ?
41                 PairOfUnary{Zero{}, x} :
42                 let _ := x.SubStructurallySmaller(y) in
43                    PairOfUnary{Suc{x.Sub(y).DivMod(y).a},
44                                x.Sub(y).DivMod(y).b}
45 }
```

Figure 3.2: Pure function calling a lemma and returning an expression in Gobra.

`SubStructurallySmaller` in line 23 before returning the value. `DivMod` is defined to perform the modulo operation on two `Unary` values, and the call to the lemma within the function is necessary to prove termination.

Figure 3.2 shows the Gobra version of the example in Figure 3.1. It features the additional struct (lines 33 to 35) that is necessary to return multiple values in a pure function and the let binding (lines 42 to 44).

3.3 Chapter 8: Sorting

The main challenge of this chapter was that Gobra could not prove termination for a method, even though we knew that it always terminates. Despite trying different termination measures, Gobra returned the following error message: *Function might not terminate. Termination measure might not decrease or might not be bounded*. We came to the conclusion that the ADT implementation still needed some further improvement to allow a more precise mapping of the `len` function, and we decided to assume termination for this method. Figure 3.3 shows the method with the termination measure that we added. Figure 3.4 provides the same code snippet in Dafny. Unlike Gobra, Dafny is able to prove termination.

```
ghost
requires length >= 0
requires length == xs.Length()
ensures xs.MergeSortAux(length).Ordered()
decreases len(xs) + length
func (xs List) MergeSortAuxOrdered(length int) {
    if length < 2 {
        xs.Split(length/2).left.MergeSortAuxOrdered(length/2)
        xs.Split(length/2).right.MergeSortAuxOrdered(length-length/2)
    } else {
        xs.Split(length/2).left.MergeSortAuxOrdered(length/2)
        xs.Split(length/2).right.MergeSortAuxOrdered(length-length/2)
        xs.Split(length/2).left.MergeSortAux(length/2).
            MergeOrdered(xs.Split(length/2).right.
            MergeSortAux(length-length/2))
    }
}
```

Figure 3.3: Gobra is not able to prove termination for `MergeSortAuxOrdered`.

```
lemma MergeSortAuxOrdered(xs: List<int>, len: nat)
    requires len == Length(xs)
    ensures Ordered(MergeSortAux(xs, len))
    decreases len
{
    if 2 <= len {
        var (left, right) := Split(xs, len/2);
        MergeOrdered(MergeSortAux(left, len/2),
                     MergeSortAux(right, len - len/2));
    }
}
```

Figure 3.4: Dafny can prove termination of `MergeSortAuxOrdered`.

3.4 Chapter 10: Data-Structure Invariants

In this chapter we needed to find a translation for modules. We decided to translate them to a package in Gobra, since it is also a construct used to encapsulate definitions. We translated definitions that were included in the export set to public functions and methods, and definitions that were not added to the set to private

```
ghost
requires pq.Valid()
ensures pq.Insert(y).Valid() && (pq.Insert(y).Elements() ==
            pq.Elements() union mset[int]{y})
decreases len(pq)
func (pq PQueue) InsertCorrect(y int) {
    match pq {
        case Leaf{}:

        case Node{?x, ?left, ?right}:
            if y < x {
                right.InsertCorrect(x)
            } else {
                right.InsertCorrect(y)
            }
    }
}
```

Figure 3.5: Ghost function that proves the correctness of the `Insert` function in Gobra.

```
ghost
requires pq.Valid()
ensures pq.Insert(y).Valid() && (pq.Insert(y).Elements() ==
            pq.Elements() union mset[int]{y})
ensures res
decreases len(pq)
pure func (pq PQueue) InsertCorrect(y int) (res bool) {
    return pq == Leaf{} ||
            (y < pq.x ?
                    pq.right.InsertCorrect(pq.x) :
                    pq.right.InsertCorrect(y))
}
```

Figure 3.6: Pure ghost function that proves the correctness of the `Insert` function in Gobra.

ones. Nonetheless, we had to make compromises in terms of accuracy because Gobra does not differentiate between public functions that reveal their entire implementation and public functions that only provide the functionality. Dafny uses the `reveals` and `provides` keywords for this.

Another challenge we encountered in this chapter was that the lemmas that we translated to pure ghost functions in Gobra could not be verified. Figure 3.5 presents one of the lemmas that we translated to a pure ghost function. The lemma states that if an element is inserted into a valid Priority Queue, the resulting Priority Queue will contain the same elements as the original one combined with the inserted element. Although we were really confident that our implementation was correct, Gobra was not able to verify it. Interestingly, the non-pure translation of the lemma, which is shown in Figure 3.6, was verified by Gobra. We believe that it is a limitation of Gobra and still requires further improvement.

3.5 Chapters 11 & 12: Loops and Recursive Specifications, Iterative Programs

In these two chapters we were introduced to methods that were using while loops in Dafny. We translated the while loops to for loops in Gobra. The two loops are semantically equivalent if all loop variables are initialized to the same value and the loop condition as well as the update rule are the same. Figure 3.7 exemplifies how the integer square root of a non-negative number can be computed using a while loop in Dafny, while Figure 3.8 shows the corresponding code in Gobra. Overall, the translations were straightforward, and we encountered no major challenges.

```
1  method SquareRoot(N: nat) returns (r: nat)
2      ensures r*r <= N < (r+1)*(r+1)
3  {
4      r := 0;
5      var s := 1;
6      while s <= N
7          invariant r*r <= N
8          invariant s == (r+1)*(r+1)
9      {
10         s := s + 2*r + 3;
11         r := r + 1;
12     }
13 }
```

Figure 3.7: Dafny method that computes the integer square root.

```
1  requires N >= 0
2  ensures r >= 0
3  ensures r*r <= N && N < (r+1)*(r+1)
4  decreases
5  func SquareRoot(N int) (r int) {
6      r = 0
7      s := 1
8      invariant r*r <= N
9      invariant s == (r+1)*(r+1)
10     decreases N-r
11         for ; s <= N; {
12         s = s + 2*r + 3
13         r = r + 1
14         }
15     return r
16 }
```

Figure 3.8: Gobra function that computes the integer square root.

3.6 Chapters 13 & 14: Arrays and Searching and Modifying Arrays

In these two chapters we encountered methods in Dafny that take arrays as arguments. Since arrays are mutable data structures in Dafny, the methods require additional annotation in order for Dafny to set up a frame. The annotation

should indicate whether the method reads from the array or writes to it. Figure 3.9 exemplifies a method in Dafny that is annotated with `modifies a`, where `a` is an array. This is required because, in line 11, the method is writing to the array.

```
1  method InitArray<T>(a: array<T>, d: T)
2      modifies a
3      ensures forall i :: 0 <= i < a.Length ==> a[i] == d
4  {
5      var n := 0;
6      while n != a.Length
7          invariant 0 <= n <= a.Length
8          invariant forall i :: 0 <= i < n ==> a[i] == d
9      {
10         a[n] := d;
11         n := n + 1;
12     }
13 }
```

Figure 3.9: Dafny method that initializes an array to a certain value.

Arrays are translated to slices in Gobra, which are also mutable data structures. We translated methods that are annotated with `modifies a` to functions in Gobra that request full permission to access all of the memory addresses of `a`. Figure 3.10 presents the translated version of Figure 3.9. Lines 1-3 show how we could specify full access permissions to the memory locations using `quantified permissions`. By using the `forall` quantifier, we were able to request permission for every address of the slice's elements, even though the concrete size of the argument was unknown.

```
1  requires forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
2  ensures forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
3  ensures forall i int :: 0 <= i && i < len(a) ==> a[i] == d
4  decreases
5  func InitArray(a []int, d int) {
6      n := 0

7      invariant 0 <= n && n <= len(a)
8      invariant forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
9      invariant forall i int :: 0 <= i && i < n ==> a[i] == d
10     decreases len(a) - n
11     for ; n != len(a); {
12         a[n] = d
13         n = n + 1
14     }
15 }
```

Figure 3.10: Gobra function that initializes a slice to a certain value.

Although the specification style of Dafny looks simpler, the one used in Gobra has its advantages. For example, in Gobra we can pass permission to only one element of a slice in order to modify it, whereas in Dafny we need to transfer permissions to the entire array using the `modifies` clause, and then add a postcondition that states what remained the same.

```
requires forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
ensures forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
ensures forall i int :: 0 <= i && i < len(a) ==>
        a[i] == old(a[(i + 1) \% len(a)])
decreases
func RotateLeft(a []int) {
    if len(a) <= 1 {
        return
    }
    n := 1
    head := a[0]

    invariant 1 <= n && n <= len(a)
    invariant forall i int :: 0 <= i && i < len(a) ==>
            acc(&a[i])
    invariant forall i int :: 0 <= i && i < n - 1 ==>
            a[i] == old(a[i + 1])
    invariant forall i int :: n <= i && i < len(a) ==>
            a[i] == old(a[i])
    decreases len(a) - n
    for ; n < len(a); {
        a[n - 1] = a[n]
        n = n + 1
    }
    a[len(a) - 1] = head

    assert forall i int :: 0 <= i && i < len(a) - 1 ==>
            a[i] == old(a[(i + 1)])
    assert len(a) > 0 ==> a[len(a) - 1] == old(a[0])
    assert forall i int :: 0 <= i && i < len(a) - 1 ==>
            (i + 1) \% len(a) == i + 1
    assert ((len(a) - 1) + 1) \% len(a) == 0
    assert forall i int :: 0 <= i && i < len(a) ==>
            a[i] == old(a[(i + 1) \% len(a)])
}
```

Figure 3.11: Gobra function that rotates every element of a slice to the left.

Some of the examples that we translated to Gobra took a very long time to be verified by Gobra. For example, Figure 3.11 ran for 10225 seconds on a Mac mini 2020 with 16GB RAM. It shows a function called `RotateLeft`, which takes a slice as an argument and rotates every element to the left. Initially, we were not sure whether Gobra was stuck, or whether it was just taking more time than expected.

Lastly, we needed to find a translation for Dafny's sequences. In Dafny, a sequence is an immutable data structure that represents an ordered collection of elements of the same type. Due to the fact that a similar data structure is not supported in Go, we decided to use slices. Additionally, each function or method that takes a slice as an argument is only allowed to use fractional permission to access the heap location, which ensures that the slice can never be modified. The element of a slice can only be changed by creating a new slice with the appropriate values. Rather than assigning a slice directly to a variable, we generated a copy and assigned it. This ensures that the variable points to a new slice with the same values, which models the immutability property of the sequence in Dafny.

An alternative approach could have been to use Gobra's sequences instead of Go's slices, as they are very similar to sequences in Dafny. However, this would have required us to declare the code as ghost.

Chapter 4

Evaluation

With the availability of examples and exercises in both Dafny and Gobra, we have been able to devise a comparison between the two languages. This chapter compares the programming capabilities of the underlying language, as well as the expressiveness of the specification and verification features. Additionally, we point out the features that were missing in Gobra and thus made the proof cumbersome or even impossible. At the end of the chapter we compare the IDE of Dafny against the one of Gobra to identify useful features that could be implemented in the Gobra IDE as well.

4.1 Gobra vs Dafny

4.1.1 Inconsistency in the Ghost Code

Gobra and Dafny both define their notions of ghost code in a similar way. However, one significant difference between them is that Gobra provides features that are not available in Go, such as ADTs and sequences. As a consequence, the functions and methods that use these concepts must be ghost, even though compilable code would have been preferred. Dafny, on the other hand, does not have this limitation, because the specification language and implementation languages are the same and evolved in tandem, unlike Gobra which proposes an extension of the Go syntax which is not supported by the compiler.

4.1.2 Correspondence Between Functions, Methods and Lemmas

During our translations we found that the following correspondences between declarations in Dafny and declarations in Gobra hold.

- A function in Dafny is ghost by default and it does not have any side effects because it models a function in a mathematical sense. This makes it equivalent to a pure ghost function in Gobra. Defining it as pure is necessary,

because it guarantees that the function can not have any side effects and thus, can be used in specifications and proof annotations.

- The function method that Dafny provides is the compilable version of the function, in that it is not ghost. The translation of a function method to Gobra would be a pure function.
- Methods in Dafny are equivalent to functions or methods in Gobra, because they are both allowed to have side effects and they are both compilable.
- Dafny's lemmas can be thought of as pure ghost functions in Gobra. It is necessary to define it as pure because this allows them to be called by other pure functions. However, lemmas that might modify the view on predicates must be translated to ghost functions in Gobra.

4.1.3 Differences in Annotation Overhead

During the translations we discovered that Dafny has various features to reduce the annotation overhead. This section summarizes the specification and verification features of Dafny that seem to be more expressive than the translated version in Gobra.

Starting with termination, Dafny always checks for it by default and tries to infer the correct termination measure automatically. Consequently, we do not have to add any lines of code unless Dafny is unsuccessful in doing so, which was very rare in the examples we worked with. This is different from Gobra, which does not check for termination if we do not explicitly require it. From our experience, finding the termination measure itself was mostly straightforward. However, adding the `decreases` clause can easily be forgotten because Gobra does not complain about non-termination.

Dafny also seems to be more efficient in terms of annotation required to set up a frame. While Dafny only requires annotation that specifies whether the method reads from the mutable data structure or writes to it, functions in Gobra need to specify precisely which access permissions are required for each memory location of the data structure. Figure 4.1 exemplifies the annotation required to initialize a two dimensional array in Dafny, and Figure 4.2 shows the specification required to initialize a two dimensional slice in Gobra.

As a last feature that can save annotation overhead, we would like to mention the automatic induction tool of Dafny that is able to prove most of the lemmas automatically. This feature is not available in Gobra and therefore we had to do the induction proofs manually, resulting in many lines of code. Not only does automatic induction reduce overhead, but it also eliminates the need to worry about how to do the proof. All induction proofs follow a similar pattern where the cases have to be split up and the lemmas can be applied recursively to smaller parts. This is a lot of work that can be avoided.

```
method InitMatrix<T>(a: array2<T>, d: T)
    modifies a
    ensures forall i, j ::
        0 <= i < a.Length0 && 0 <= j < a.Length1 ==> a[i, j] == d
{
    var m := 0;
    while m != a.Length0
        invariant 0 <= m <= a.Length0
        invariant forall i, j ::
            0 <= i < m && 0 <= j < a.Length1 ==>
                a[i, j] == d
    {
        var n := 0;
        while n != a.Length1
            invariant 0 <= n <= a.Length1
            invariant forall i, j ::
                0 <= i < m && 0 <= j < a.Length1 ==> a[i, j] == d
            invariant forall j :: 0 <= j < n ==> a[m, j] == d
            {
                a[m, n] := d;
                n := n + 1;
            }
        m := m + 1;
    }
}
```

Figure 4.1: Initializing a two dimensional array to a certain value in Dafny.

```
requires forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
requires forall i, j int :: 0 <= i && i < len(a) &&
                        0 <= j && j < len(a[i]) ==> acc(&a[i][j])
ensures forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
ensures forall i, j int :: 0 <= i && i < len(a) &&
                        0 <= j && j < len(a[i]) ==> acc(&a[i][j])
ensures forall i, j int :: 0 <= i && i < len(a) &&
                        0 <= j && j  < len(a[i]) ==> a[i][j] == d
decreases
func InitMatrix(a [][]int, d int) {
    m := 0

    invariant 0 <= m && m <= len(a)
    invariant forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
    invariant forall i, j int :: 0 <= i && i < len(a) &&
                            0 <= j && j < len(a[i]) ==> acc(&a[i][j])
    invariant forall i, j int :: 0 <= i && i < m &&
                            0 <= j && j < len(a[i]) ==> a[i][j] == d
    decreases len(a) - m
    for ; m != len(a); {
        n := 0

        invariant forall i int :: 0 <= i && i < len(a) ==> acc(&a[i])
        invariant forall i, j int :: 0 <= i && i < len(a) &&
                                0 <= j && j < len(a[i]) ==> acc(&a[i][j])
        invariant 0 <= n && n <= len(a[m])
        invariant forall i, j int :: 0 <= i && i < m &&
                                0 <= j && j < len(a[i]) ==> a[i][j] == d
        invariant forall j int :: 0 <= j && j < n ==> a[m][j] == d
        decreases len(a[m]) - n
        for ; n != len(a[m]); {
            a[m][n] = d
            n = n + 1
        }
        m = m + 1
    }
}
```

Figure 4.2: Initializing a two dimensional slice to a certain value in Gobra.

To compare the annotation overhead of Gobra and Dafny, we created Table 4.1. We defined the annotation overhead to be the number of lines of non-compilable code divided by the number of lines of code and multiplied by 100. Non-compilable code includes specifications, annotations, assert statements and other ghost code. There is one exception. Gobra's ghost functions that have to be defined as ghost because they are using constructs, like ADTs, that are only available in the extended Gobra language, are counted as compilable code.

Chapter	Dafny Overhead	Gobra Overhead	Difference (Dafny-Gobra)
6	74 %	82 %	-8 %
7	69 %	80 %	-11 %
8	57 %	75 %	-18 %
10	56 %	62 %	-6 %
11	34 %	45 %	-11 %
12	46 %	57 %	-11 %
13	46 %	63 %	-17 %
14	40 %	58 %	-18 %
Average	53 %	65 %	-13 %

Table 4.1: Compares the annotation overhead per chapter between Dafny and Gobra based on the examples.

Our initial feeling that Dafny is less verbose than Gobra was confirmed by our measurements in Table 4.1. Dafny requires approximately 13% less annotation overhead on average.

To evaluate the effectiveness of Dafny's automatic induction tool, we compared the number of lines of code needed to prove a lemma in Dafny with and without automatic induction. The comparison was based on all lemmas that were available in both versions. We found that using automatic induction can significantly reduce the amount of code the programmer needs to provide. The overhead in the number of lines of code can be reduced by around 78% according to our measurements.

4.1.4 Missing or Incomplete Features

There were only a few examples of features in Gobra that were missing or not as advanced as Dafny's, and made the translation cumbersome or even impossible. Gobra's ADTs, for example, lack support for generic data types, which made it impossible to translate the ADTs from Dafny accurately. As already mentioned, we had to make a trade-off and decided to use `int` instead.

4.2 Gobra IDE vs Dafny IDE

We compared the Gobra IDE to the Dafny IDE in the hope to find useful features that could be implemented to the Gobra IDE as well. In this section, we introduce two features currently available only in the Dafny IDE that could improve the programming experience in Gobra.

Features only Supported by the Dafny IDE

Displaying Counter proof

The Dafny IDE has a feature that displays a counter example for a failing proof. If the proof fails, programmers can request Dafny to return a set of values that lead to an incorrect result. This information can be very useful to the programmer to identify the source of the problem. We believe that its implementation in the Gobra IDE could be very beneficial for the users of Gobra, as it would improve the debugging experience. There is already some progress on this front in the form of a bachelor's thesis.

Go to Definitions

Another feature that is only offered by the Dafny IDE is to jump to the corresponding definitions. This feature can be helpful for programmers that encounter a variable or a function within the code and want to learn more about its implementation. It can save a lot of time, especially when working on a larger project.

Chapter 5

Conclusion

The main goal of our thesis was to provide well-documented introductory examples and exercises to Go programmers interested in learning program verification. During our work, we successfully translated numerous examples and exercises from Dafny to Gobra with high accuracy. Some of them were translated in an idiomatic way and others in a direct one. We documented the code well to ensure its usefulness as teaching material. Additionally, we devised a comprehensive comparison between Dafny and Gobra based on the book's examples and exercises. Finally, we also compared the IDE of Dafny with the one of Gobra and identified that it would be beneficial to implement a feature that displays a counter example for a failing proof and a feature that allows to quickly navigate to the corresponding definition.

Due to the lack of time, we were only able to translate one additional chapter as part of our extension goals. We also did not manage to implement some of the features in Gobra already, but they may be addressed in future work.

It is fair to say that writing code in Dafny is currently more convenient for the programmer. Not only is it more concise, but it also supports the programmer with features, like automatic induction, that can save a lot of time and effort. However, it is amazing to see that we can translate almost everything to Gobra with almost no compromises. We believe that Gobra is an excellent verifier for the Go programming language, and with the ongoing improvements, it is becoming even better. We also believe that it would be beneficial to provide even more introductory material, such as an interactive tutorial or a book in the style of Program Proofs.

Future Work

With this section we would like to keep track of some tasks that could be done in the future to improve Gobra as well as its teaching materials even further.

We plan to add support for polymorphism in the ADT implementation of Gobra. Once this has been implemented, we can revisit the examples and exercises in chapter 6 of the book and improve the translation to also use generic data types.

Due to the fact that the ADT implementation needed further improvement, we had to assume termination in chapter 8. Our plan is to improve the implementation of the ADT to a point where termination can be proven. Assuming that the ADT has been improved, we can revisit chapter 8 and change the termination measure to a correct one.

We found the automatic induction of Dafny to be very useful. Our plan is to add this feature to Gobra as well. Additionally, we also want to implement the feature that displays a counter example and the feature that allows to quickly navigate to the corresponding definition in the Gobra IDE.

Bibliography

[1] Dafny 4 is released. https://dafny.org/blog/2023/03/03/dafny-4-released. [Online], Accessed on 2023-04-07.

[2] Dafny reference manual. https://dafny.org/dafny/DafnyRef/DafnyRef. [Online], Accessed on 2023-04-07.

[3] Gobra: Modular specification and verification of go programs. https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17. [Online], Accessed on 2023-04-07.

[4] A tutorial on gobra. https://github.com/viperproject/gobra/blob/master/docs/tutorial.md. [Online], Accessed on 2023-04-07.

[5] Paul Dahlke. Extending the specification language of a go verifier with algebraic data types. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Paul_Dahlke_BA_Report.pdf. [Online], Accessed on 2023-04-07.

[6] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. https://www.andrew.cmu.edu/user/bparno/papers/ironfleet.pdf. [Online], Accessed on 2023-04-07.

[7] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel. [Online], Accessed on 2023-04-07.

[8] I. T. Kassios. The dynamic frames theory. https://pm.inf.ethz.ch/publications/Kassios10.pdf. [Online], Accessed on 2023-04-07.

[9] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. https://www.microsoft.com/en-us/research/wp-content/uploads/2008/12/dafny_krml203.pdf. [Online], Accessed on 2023-04-07.

[10] K. Rustan M. Leino. Program proofs. https://mitpress.mit.edu/9780262546232/program-proofs/. [Online], Accessed on 2023-04-07.

[11] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. https://dl.acm.org/doi/pdf/10.1145/2160910.2160911. [Online], Accessed on 2023-04-07.

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):

Translating Pedagogical Exercises to Viper's Go Front-end

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Egli	Timon

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	**Signature(s)**
Grüningen, 09. April 2023	T. Egli

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