Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Standard Library for Gobra

Practical Work

Daniel Nezamabadi

April 01, 2024

Advisors: João Carlos Mendes Pereira, Prof. Dr. Peter Müller
Department of Computer Science, ETH Zürich

Abstract

Gobra is an automated verifier for Go programs based on separation logic. We implement a standard library containing useful definitions and lemmas used in specifications and proofs to provide code reusability across projects and hopefully allow improvements in performance and proof stability. Additionally, we extend Gobra by allowing the user to turn off set axiomatization, proving any proof obligations manually using the standard library, and by adding the keywords `opaque` and `reveal`, which give fine-grained control to the user over when the body of a function is revealed. We evaluate the effect of `opaque`, turning off set axiomatization and "assisting" the verifier with additional assertions and instantiations of lemmas from the standard library on quantifier instantiations and execution time. While using `opaque` decreases the number of quantifier instantiations, execution times do not. This may be due to the relatively small scale of the examples tested. In general, it appears that the number of quantifier instantiations does not correlate with execution time. Disabling set axiomatization and manually proving the proof obligations results in a decrease in quantifier instantiations and execution time. While the effect of assisting the verifier on quantifier instantiations is unclear, the data suggests that in small examples verified with a symbolic executing engine, not assisting the verifier at all results in the best performance in terms of execution time.

Contents

Chapter 1

Introduction

A standard library is an essential component of any major programming language. It provides every programmer with functionality whose implementation, verification, optimization, and maintenance do not need to be duplicated across projects.

Standard libraries are not limited to conventional programming languages: program verifiers like Dafny [2], Verus [10], and Why3 [24] also provide standard libraries that contain, among other things, commonly used lemmas and definitions. Gobra [12], a verifier for Go programs [14] used to verify real-world projects like SCION [21] and a part of the official Go implementation of WireGuard [13], does not yet have a standard library.

In addition to minimizing repetition across projects, developing a standard library could enable improvements in performance and proof stability, by allowing more fine-grained control over the proof context, the set of facts available to the verifier to prove the postconditions of a function. Two mechanisms could achieve this: firstly, by reducing the context, where unhelpful facts are minimized to prevent the verifier from going off-track, which could lead to instability and performance issues. This reduction can be realized by allowing the user to selectively turn off features of the SMT solver, requiring programmers to leverage the standard library to manually prove proof obligations that depend on the disabled features. These problematic features include non-linear integer arithmetic and axiomatization of built-in data structures like lists and sets. Another way to reduce the context is by utilizing the keyword `opaque`, which hides function bodies unless explicitly revealed by the programmer.

We used ChatGPT 3.5/4 for language tasks like getting suggestions for headlines and glueing together sentences. Additionally, these tools and GitHub Copilot were used to assist in writing the Python and Bash scripts for the evaluation [7].

Secondly, guiding the verifier by instantiating lemmas from the standard library may lead the verifier faster and more reliably to a solution. For example, it may be useful to explicitly use lemmas that a person would use to prove the postconditions.

During this practical work, we have begun to lay the foundation of investigating these ideas in more detail by implementing an initial set of packages inspired by the standard libraries of the verifiers mentioned above, introducing an option to turn off set axiomatization and implementing as well as evaluating the keyword `opaque`.

In this report, we provide an overview of our work and evaluate the impact of reducing the verifier's context and guiding the verifier by instantiating lemmas from the standard on the number of quantifier instantiations and overall performance. We also highlight challenges encountered in our experiences working with Gobra. The report concludes with a summary of our efforts and suggests potential areas for further development to improve Gobra.

1.1 Background

1.1.1 Gobra

Gobra is an automated verifier for Go programs based on separation logic. It provides numerous features to verify a wide range of properties, including memory safety, crash safety, data-race freedom, and partial correctness, based on user-provided specifications.

Firstly, functions and methods can be annotated with pre- and postconditions, which are verified modularly. This means that a call only uses the specification of the callee and cannot peek inside the body. If a function or method has no side effects, such as modifying heap-allocated structures, and is deterministic, it can be annotated with `pure`, allowing the usage in specifications. Pure functions and methods have an implicit postcondition that ensures their result is equal to their body.

```
ensures res == (n % 2 == 0)
pure func isEven(n int) (res bool) {
  return n % 2 == 0
}

ensures isEven(n) ==> res == n / 2
ensures !isEven(n) ==> res == n / 2 + 1
func halfRoundedUp(n int) (res int) {
  if isEven(n) {
    res = n / 2
  } else {
    res = n / 2 + 1
```

```
    }
    return res
}
```

Listing 1.1: Examples of annotated functions, as seen in the Gobra Tutorial [22]

Reasoning about the heap is enabled with the concept of access permissions. Access permissions are held by method executions. To read from a heap location `x`, the method must hold any non-zero amount of the access permission `acc(x)`. To write to a heap location, the method must hold the entire access permission associated with the heap location. Access permissions are transferred between methods upon call and return.

Gobra also supports predicates, which can abstract away invariants for non-trivial data structures like slices or recursively defined lists.

```
type node struct {
  value int
  next *node
}

pred list(ptr *node) {
  acc(&ptr.value) && acc(&ptr.next) &&
  (ptr.next != nil ==> list(ptr.next))
}
```

Listing 1.2: Recursive list using predicates and access permissions, as seen in the Gobra Tutorial [22]

Sometimes, additional code is needed to verify 'real' code. This is referred to as ghost code. Its main characteristic is that it is not compiled, meaning that the control flow of the 'actual' program cannot depend on it – a property automatically checked by Gobra.

Gobra implements ghost variants for various constructs: in and out parameters, variables, types, statements, methods, and functions.

In addition to allowing ghost code, Gobra provides many ghost types: sequences, sets, multisets, and the `Perm` type for permission amounts. It also provides common operations on these ghost types, e.g., sequence concatenation, set union, membership, multiplicity, sequence length and set cardinality. Using domains, it is also possible to define additional types through mathematical functions and axioms providing their properties. Gobra also supports the definition of Algebraic Data Types (ADTs), which are implemented in many programming languages, including Haskell and Rust. By default, Gobra instantiates axioms for ghost types wherever they are relevant. As mentioned above, these instantiations may cause proof instability and performance degradation.

```
type List adt {
  Nil{}
```

```
    Cons{int List}
}
```

Listing 1.3: List definition using an ADT

For a more in-depth discussion of Gobra's features, we refer to the Gobra and Viper tutorial [22, 20].

1.1.2 Dafny

Dafny is an imperative, sequential, verification-aware programming language. It supports common programming concepts like inductive data types (ADTs), subset types (e.g., bounded integers), lambda expressions and functional programming idioms, and immutable and mutable data structures. Being verification-aware, Dafny also supports program verification concepts like quantifiers, calculational proofs (stepwise formula manipulation), the ability to use and prove lemmas, pre- and postconditions, termination conditions, loop invariants and read/write specifications. `opaque` is also present in Dafny, which, as described above, allows developers to define functions and predicates whose body is hidden, except in places where the body is explicitly revealed using `reveal`. This allows more abstract reasoning and control over the proof context, i.e., the "facts" available to the verifier during verification.

1.1.3 Verus

Verus is a tool designed for verifying the correctness of code written in Rust, contrasting with Dafny, which is a programming language. Like Gobra, Verus focuses on verifying the correctness of code in an existing, "real-world" language. Specifically, Verus' standard library, which is inspired by Dafny's standard library and axioms, serves as a useful reference for our work.

1.1.4 Why3

Why3 is a platform for deductive program verification. It provides a language for specification and programming called WhyML and relies on external provers, both automatic (e.g., Alt-Ergo, CVC4, Z3) and interactive (e.g., Coq, PVS, Isabelle/HOL), to discharge verification conditions. Gobra and Dafny do not provide this level of flexibility.

1.1.5 Dafny and Why3's Standard Library

Much of the code in Dafny's core library [3] comes from or was inspired by code from verification projects like Ironclad [23], Vale [15], Verified BetrFS [18] and Verifying OpenTitan [16]. It provides, among other things, definitions of types, operations, and lemmas for a wide range of modules: wrappers (e.g., `Option<T>`), bounded integers, basic file I/O, basic mathematical operations

and lemmas, Unicode, collections (e.g., sequences, sets, maps), non-linear arithmetic (NLA), relations (i.e., properties of functions) and properties of binary operations.

```
lemma LemmaMulBasics(x: int)
  ensures 0 * x == 0
  ensures x * 0 == 0
  ensures 1 * x == x
  ensures x * 1 == x
{
}
```

Listing 1.4: Lemma for NLA from the Dafny Core Library [1]

Most of these concepts can be found in some form in Why3's standard library [25] as well. On top of those, Why3 implements a wide range of other theories like IEEE floats, graph theory, and hash tables, to name a few.

Chapter 2

Overview

This chapter gives a brief overview of what we did throughout this practical work and our reasons for doing so.

Disabling Set Axiomatization We extended Gobra to allow the axiomatization of sets to be disabled by passing an empty axiomatization file to Silicon. A user of this option must manually prove all proof obligations. While this comes with annotation overhead, it may allow for good performance and stable proofs. To reduce the burden on the user, we also implement a standard library containing useful definitions and lemmas, which we will elaborate on later.

`opaque` **and** `reveal` We added the keyword `opaque` to Gobra, which can be used on pure functions to hide their bodies and only reveal them for specific function calls if explicitly requested using `reveal`. This allows for more fine-grained control over what is part of the verifier context, which in turn may translate into improved control over performance and proof stability. `opaque` is not implemented for non-pure functions, as these do not expose their bodies.

Standard Library We implemented packages that contain definitions and lemmas commonly used in specifications. These packages are for mathematical and Go's maps, sequences, sets, basic math definitions and commonly used utilities for verification. We give an example of one such lemma in Listing 2.1.

```
// Subset relation is transitive.
ghost
opaque
requires xs subset ys
requires ys subset zs
ensures xs subset zs
```

```
decreases
pure func SubsetIsTransitive(xs, ys, zs set[int]) util.Unit {
    return util.Unit{}
}
```

Listing 2.1: Subset transitivity lemma in the standard library

The packages' definitions were heavily inspired by Dafny, Why3, and Verus, the latter of which also appears to be inspired by Dafny. While some definitions were straightforward to port, others required more effort, as Dafny, in particular, is at times more expressive than Gobra.

We implement a standard library for broadly two reasons, the first of which is minimizing repetition across projects. This has the typical benefits from a software engineering perspective, as mentioned in the introduction: implementation, verification, optimization, and maintenance of functionality do not need to be duplicated if they are part of the standard library.

The second reason is that a standard library may play a part in improving performance and proof stability, for example, by disabling axiomatization and manually proving proof obligations using the standard library instead. Additionally, it allows for further exploration in this direction. One of these directions we explore is whether assisting the verifier by instantiating lemmas from the standard library improves performance and stability.

Evaluation We developed Python scripts to measure the execution time of Gobra and automatically extract the number of quantifier instantiations during the verification of a package into the CSV format and to analyze and compare the generated CSVs using box plots. Additionally, we assembled a set of interesting programs and evaluated them using these tools to investigate the effect of `opaque`, manually proving proof obligations after turning off set axiomatization and assisting the verifier using intermediate assertions and lemmas from the standard library.

Chapter 3

Evaluation

In this chapter, we want to investigate the following questions:

- What is the effect of `opaque` on the number of quantifier instantiations and execution time?
- What is the effect of turning off set axiomatization and manually proving the proof obligations on quantifier instantiations and execution time?
- What is the effect of "assisting" the verifier by asserting additional statements and instantiating lemmas on quantifier instantiations and execution time?

We will also report on Gobra's limitations that prevented us from being productive or impeded the applicability of our work.

3.1 Experiments

3.1.1 Experiment Setup

The experiments are run on a laptop with an Intel Core i5-8625U @ 1.6GHz and 8GB RAM. We extract the number of quantifier instantiations by processing output generated after enabling quantifier instantiations profiling in Z3 [4] and measure execution time as the time it takes for that command to finish. The following versions of the programs were used:

- **Silicon**: commit 0608ac9 (29.02.2024) [8]
- **Gobra**: version 0608ac92, commit da25624 (04.03.2024) [6]
- **Z3**: version 4.8.7 (19.11.2019) [9]

Silicon is one of Gobra's verification backends and uses Z3. We use an old version of Z3, as enabling quantifier instantiation profiling in newer versions

results in errors and output for which we could not find any documentation [5].

All experiments were run for 30 iterations. We plot the data as boxplots: a box corresponds to the three quartile values (25th percentile, median, and 75th percentile), while the "whiskers" contain points within 1.5 of the interquartile range. The diamonds represent outliers. The axes have different scales and do not necessarily start from zero.

3.1.2 Using `opaque`

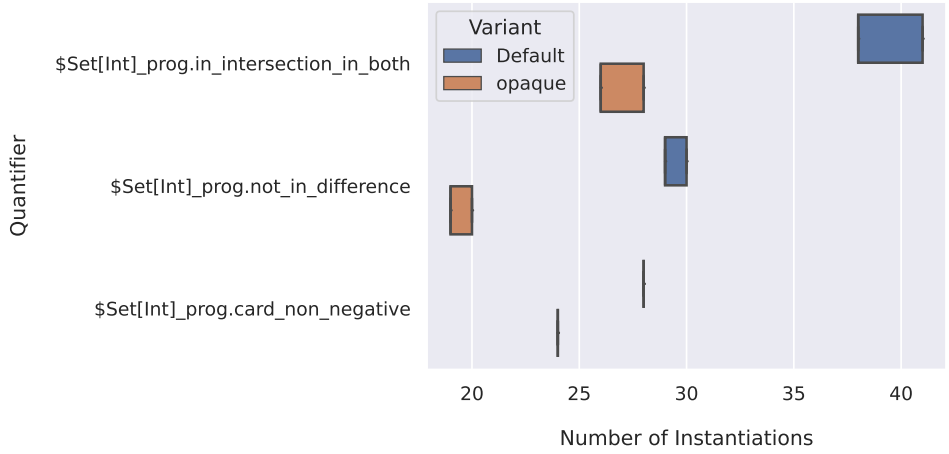
To evaluate the effect of opaque, we verify three packages with increasing complexity and compare the number of quantifier instantiations and the execution time between versions of the package where lemmas have been turned opaque or not. The first package contains a non-trivial lemma from the `sets` package of the standard library and its dependencies. The second package will be the entirety of the `sets` package. The final package we consider here will be the entirety of the `dicts` package, which uses the `sets` package.

As `opaque` hides the body of pure functions, this experiment essentially investigates the effect of hiding a lemma's proof from users. Note that lemmas proven inductively use themselves on a "smaller" instance and thus use themselves. Since the proof of a lemma is unnecessary to a user, hiding it does not require users to adapt their proofs or programs. By removing unnecessary information from the verifier's proof context, we expect quantifier instantiations and execution time in the `opaque` version of the packages to decrease.

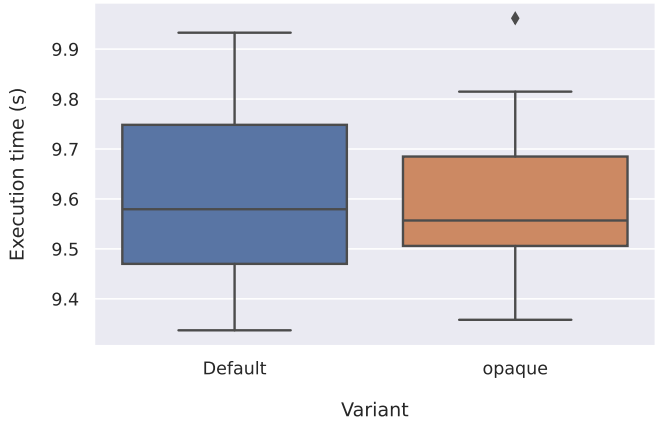
Looking at figures 3.1a, 3.2a, and 3.3a, we can see that using `opaque` consistently reduces the number of quantifier instantiations. However, if we consider figures 3.1b, 3.2b, and 3.3b, which plot the execution times, we cannot observe a clear effect of `opaque`: in Figure 3.1b median execution time and variability seem to be smaller in the `opaque` version of the package, whereas in figures 3.2b and 3.3b they seem to be the same or slightly.

This is surprising, as we expected that fewer quantifier instantiations would translate to shorter execution times. Nonetheless, these experiments do not conclusively show that `opaque` will never affect execution times. The effects of `opaque` on execution time may only reveal themselves in larger projects. This is because in larger projects more quantifiers and thus triggers are present. As `opaque` hides the function bodies, those triggers would have fewer opportunities to cause unnecessary quantifier instantiations, leading to potentially improved execution time.

Taking this into consideration, we infer that using `opaque` can be an effective way to reduce the number of quantifier instantiations and improve



(a) Top Three Quantifier Instantiations



(b) Execution Time

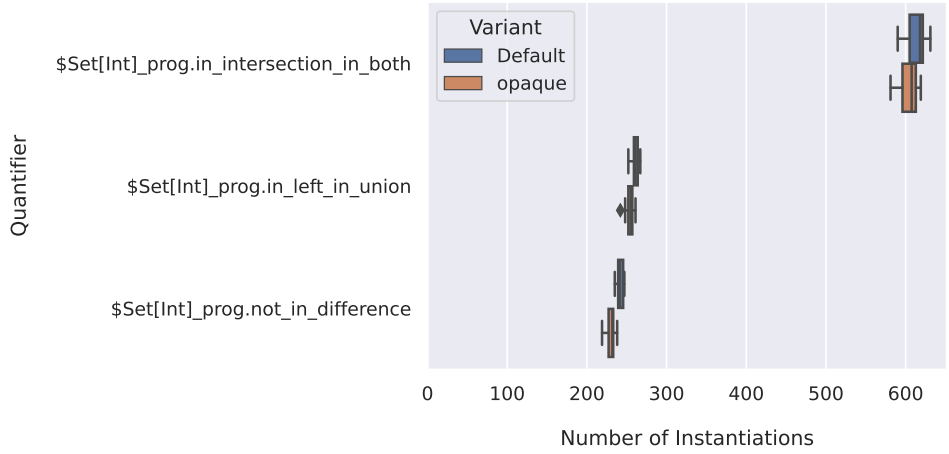
Figure 3.1: Verification of a lemma from the `sets` package

performance if said quantifier instantiations are a problem.

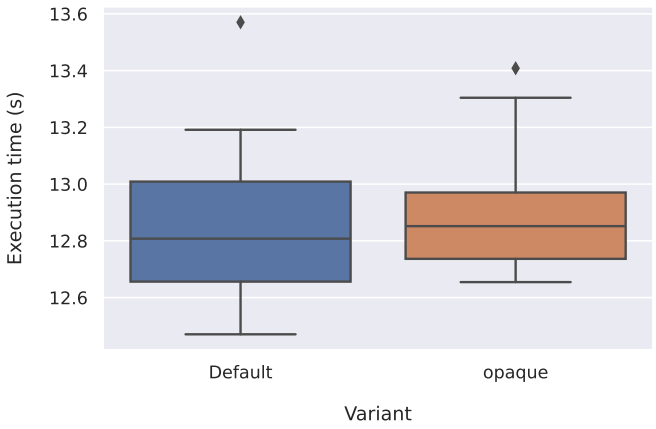
It is worth noting that in Figure 3.3a, `k!1144` is one of the quantifiers that was instantiated the most and was barely affected by `opaque`. This makes sense as it is most likely an auxiliary quantifier created by Z3 during model construction [11].

3.1.3 Manual Proofs

We will now evaluate the effect of turning off set axiomatization and manually proving the required proof obligations using the standard library on the number of quantifier instantiations and execution time. We achieve this by considering a package that has been designed to cause a large amount of



(a) Top Three Quantifier Instantiations

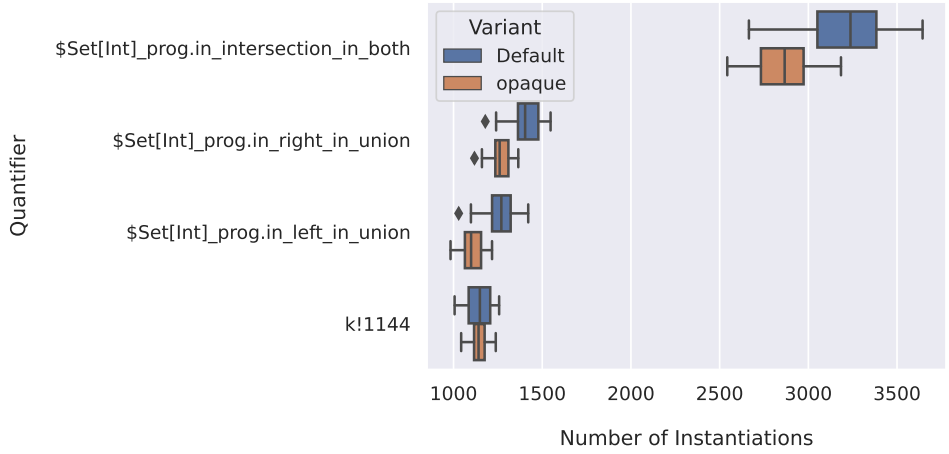


(b) Execution Time

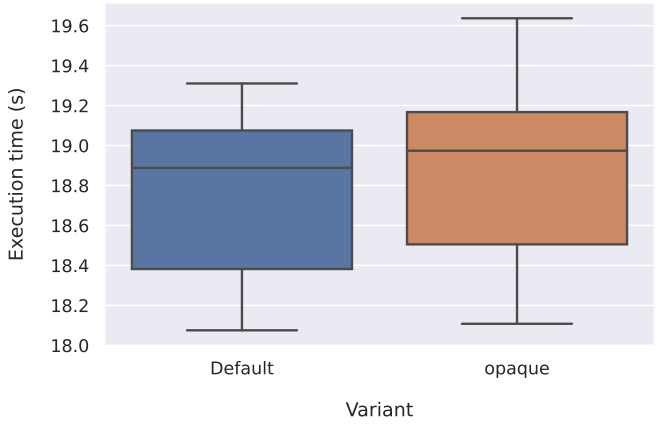
Figure 3.2: Verification of the `sets` package

quantifier instantiations. As a baseline, we will use a fully automatic version of the proof: we do not "assist" the verifier in any way; that is, we do not add additional `assert` statements to guide the verifier to a solution. We will explore this direction in the next section. An overview of the number of quantifier instantiations for the automatic version is provided in Figure 3.4.

The effect of turning off set axiomatization and manually proving the required proof obligations is significant: the only quantifier instantiations are three instantiations of `prog.getter_over_tuple2`, which is equivalent to verifying an empty file. Additionally, as shown in Figure 3.5, the median execution time of the manually proven version is reduced by 0.5s, albeit the variability seems to have increased slightly.



(a) Top Four Quantifier Instantiations



(b) Execution Time

Figure 3.3: Verification of the `dicts` package

While the reduction in quantifier instantiations aligns with our expectations, the decrease in median execution time may appear disappointing. The relatively small decrease in execution time may be due to the small scale of the example: even though the automatic verification causes quantifier instantiations in the hundreds or even thousands, the verifier may be able to easily converge to a solution. Thus, in this example, the execution time may be dominated by some verification overhead, not the problem's difficulty.

3.1.4 Assisting the Verifier

As our final set of experiments, we will investigate the effect of "assisting" the verifier by asserting additional properties and instantiating lemmas. We will

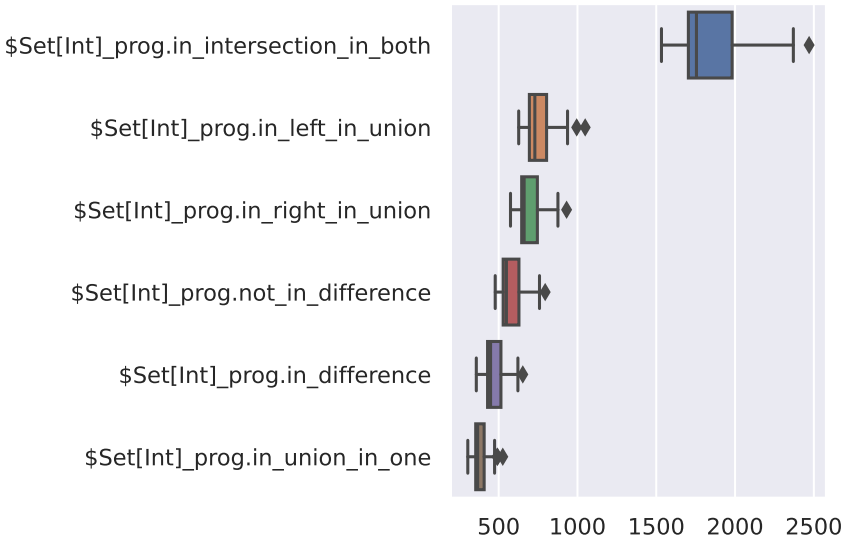


Figure 3.4: Quantifier instantiations for the automatic, unassisted synthetic function verification.

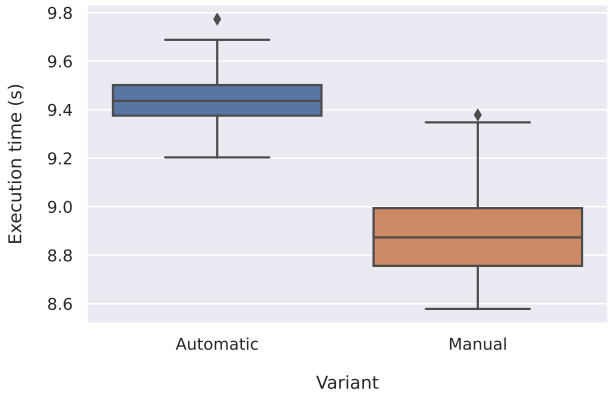


Figure 3.5: Execution time of verifying automatically and manually proven synthetic function

consider two examples: the synthetic example investigated in the previous section and a proof from chapter 10.2 of the book "Program Proofs" [19], which was ported to Gobra in the context of Timon Egli's bachelor thesis [17].

Synthetic

- **Fully Assisted**: Instantiate lemmas for every proof obligation.
- **Weak Eq**: Remove an unnecessary postcondition from a lemma.
- **No Eq**: Do not instantiate lemmas for assertions of the form $|xs - ys| = |xs| - |xs \cap ys|$
- **No Eq/Upper**: Do not instantiate lemmas for assertions of the form $|xs - ys| = |xs| - |xs \cap ys|$ or $|xs \cup ys| \leq |xs| + |ys|$
- **Unassisted**: Do not instantiate any lemmas.

We consider five different variations of the synthetic function: Fully Assisted, Weak Eq, No Eq, No Eq/Upper, and Unassisted.

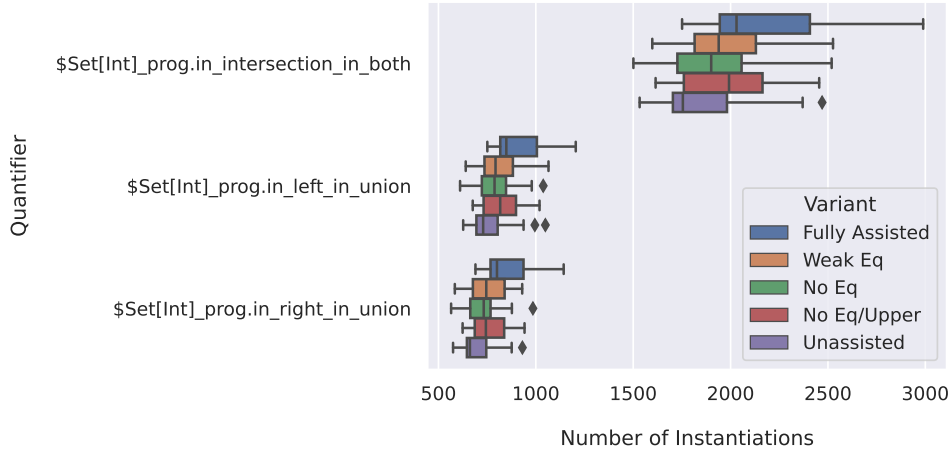
To assist the verifier, we use lemmas from the standard library, which have been copied to the package of every variant without their function bodies. We do not include the function body so as not to unnecessarily verify the lemmas again, and we include the copies in the packages of all variants, even if they are not used, to reduce the factors that may influence the measurements in case there is an overhead associated with the definition of abstract functions.

We note that going from Fully Assisted to Weak Eq does not correspond to decreased assistance. Instead, it eliminates unnecessary facts from the context by weakening the lemma from the standard library.

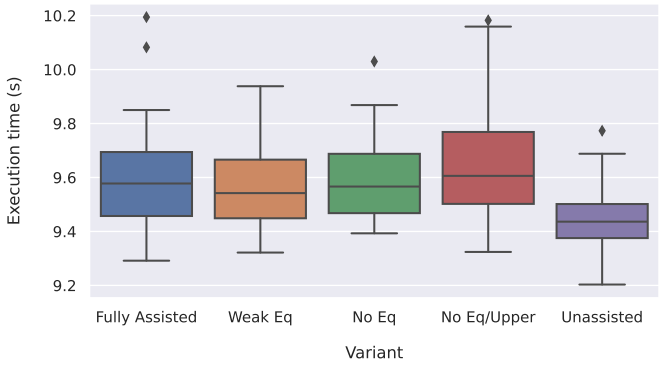
While we observe a decrease in quantifier instantiations in Figure 3.6a when removing an unnecessary postcondition (Weak Eq), execution time barely changes as seen in Figure 3.6b.

Assisting the verifier does not seem to consistently affect quantifier instantiations: for example, consider the quantifier `in_intersection_in_both`. Comparing Weak Eq to No Eq in 3.6a, we see that reducing assistance slightly decreases quantifier instantiations. If we now consider No Eq/Upper, which corresponds to a further decrease in assistance, we can observe a slight increase in quantifier instantiations. In both cases, the decrease in assistance appears to lead to a slight increase in the median execution time. Additionally, we observe that the variance of the execution time for No Eq/Upper is the largest.

Counterintuitively, not assisting the quantifier results in the least number of quantifier instantiations and execution time. In this case, the variance



(a) Top Three Quantifier Instantiations



(b) Execution Time

Figure 3.6: Verifying the synthetic function with different levels of assistance

in quantifier instantiations and execution time appears to be as good as any other level of assistance. Furthermore, the fully assisted version of the experiments seems to have the most variance in quantifier instantiations. On the one hand, it makes sense that if more things are in the verifier's context due to assistance, we have more quantifier instantiations; on the other hand, if the proof obligation is already proven by instantiating a corresponding lemma, one would expect that the variation in quantifier instantiations should be smaller and fewer quantifier instantiations are needed.

In regards to execution time, we would expect that with more assistance, execution time and its variation should decrease, but that is not the case. Execution time seems to be best if the verifier is left alone.

Program Proofs

In Chapter 10.2, we consider four variations of the proof: Full, First Half, Last Half, and Minimal.

- **Full**: Instantiates the induction hypothesis and asserts properties of the data structure.
- **First Half**: Only instantiates the induction hypothesis and properties asserted in the full version occurring before it.
- **Last Half**: Only instantiates the induction hypothesis and properties asserted in the full version occurring after it.
- **Minimal**: Only instantiates the induction hypothesis.

Note that the asserted properties are not "difficult" because they do not require any general lemma but follow relatively easily from the definitions.

Figure 3.7a plots the top five quantifier instantiations of the package, which only assists in the first half of the proof. We note the following: Firstly, the top three quantifier instantiations are auxiliary quantifiers created by Z3, and the largest number of quantifier instantiations is less than a hundred. In particular, the distribution of quantifier instantiations appears to be bimodal: for example, in different executions, we have either 30 or around 85 instantiations of the quantifier `card_non_negative`. Comparing this to the top quantifier instantiations of other versions plotted in Figure 3.7b, this is unusual: There, the number of quantifier instantiations appears to be in the hundreds with significant variability.

If quantifier instantiation were a strong predictor of execution time and its variability, we would expect execution time to be lowest in First Half and have very small variability. However, this is not the case: as seen in Figure 3.7c, while First Half executions are faster than Full or Last Half, Minimal has the lowest execution time. Additionally, the variability of the execution time of First Half is larger than Full and comparable to that of Last Half, which has the largest variation in the number of quantifier instantiations.

Looking at Figure 3.7b in more detail, we see that going from Full to Last Half makes quantifier instantiations consistently worse in both median and variability. From Full to Minimal, the median and variability of quantifier instantiations get worse in 3 cases, but in 2 cases, quantifier instantiations drop to almost 0 with barely any variability. As previously described, going from Full to First Half results in the largest reduction in quantifier instantiations. Thus, the effect of (reducing) assistance on the variability of quantifier instantiations is unclear.

This is unexpected, as one would expect that at least variability decreases with increasing assistance. These observations seem consistent with the

experiment using the synthetic function: assisting the verifier doesn't seem to have a consistent effect on quantifier instantiations, and not manually assisting the verifier seems to perform best in execution time. Finally, similarly to the previous observations, quantifier instantiations alone cannot explain execution time.

3.2 Limitations of Gobra

In this section, we give a brief overview of problems with Gobra that kept us from being productive or impeded the applicability of our work.

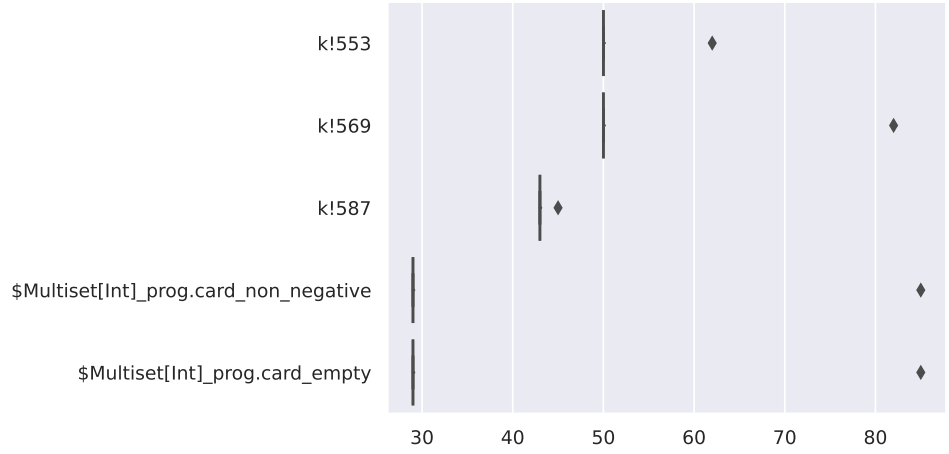
Lack of Generics At the moment, Gobra does not support generics. Consequently, the standard library contains definitions only for collections of integers, even though most of them would apply to any type. This limits the applicability of the standard library, especially in real-world projects like SCION [21].

Lack of `closed` The keyword `closed` would be similar to `opaque` in that it hides the body. However, unlike `opaque`, there would be no way to reveal the body. It would be preferable if lemmas in the standard library could be marked with `closed`, as the proofs are considered implementation details that shouldn't be exposed to users.

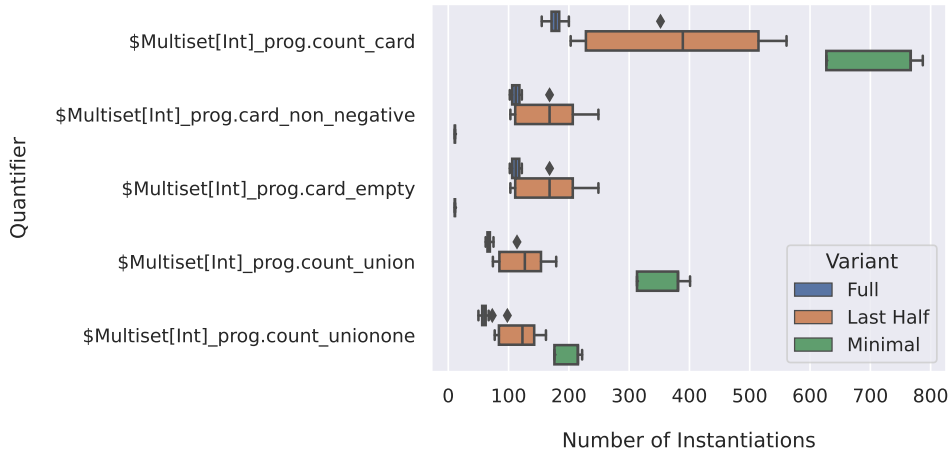
Scope Leakage Currently, Gobra does not properly respect access modifiers when importing packages. When importing, Gobra adds the entire package to the verifier's context, causing the client to try to verify private definitions or public lemmas, including `opaque` functions. In the context of the standard library, if the client disables, for example, set axiomatization, verification will fail, as they cannot verify the imported, opaque lemmas without the built-in axioms.

Restricted Return Values for Pure Functions Currently, pure function can only have one single return value. This may become a problem if the standard library's support for Go's maps, which have been suggested to be prone to more verification problems, needs to be extended, as they make significant use of multiple return values.

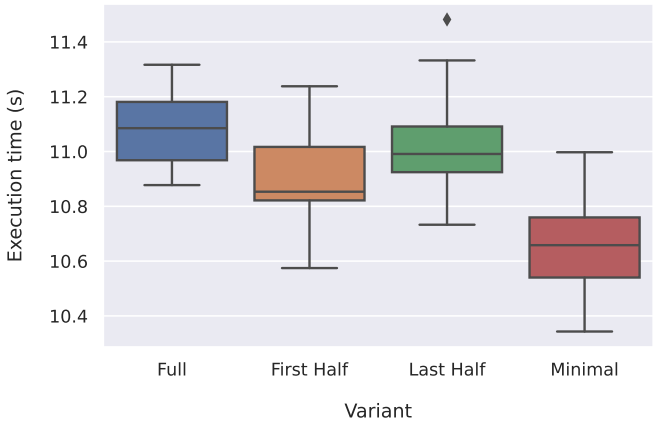
Lack of Docmentation Gobra lacks documentation in both using and extending it. The lack of overview of the architecture and classes used to implement Gobra caused a lot of time spent clicking through the code, even though adding `opaque` was not particularly complicated. Additionally, the lack of a complete user manual made it sometimes unclear what Gobra



(a) Top Five Quantifier Instantiations (only assisting with the first half)



(b) Top Five Quantifier Instantiations (without First Half)



(c) Execution Time

Figure 3.7: Verifying an example from Chapter 10.2 of Program Proofs with different levels of assistance

could do and how it could do it. While the thorough test suite was quite useful in alleviating these problems, this form of documentation is not ideal.

Editor Support Previously, the VSCode plugin for Gobra regularly hanged VSCode. This was presumably caused by Gobra trying to verify the code when the user stopped typing by default, and it was fixed.

Incompleteness Occasionally, Gobra was not able to prove assertions for unclear reasons. On one such occasion, the issue was fixed by updating to a newer version. Nonetheless, significant time was spent trying to get the assertion through by adding intermediate assertions.

Lack of Tooling At the moment, Gobra feels like a black box, both in terms of whether assertions can be proven and in terms of performance. While this may be fine if proofs go through quickly, it makes analyzing and fixing problems when they do not quite difficult.

For example, when Gobra fails to verify an assertion, it is unclear what Gobra "knows" or not. Thus, the user must guess and manually add intermediate assertions to find the problem. However, this requires them to run verification repeatedly until the problem is fixed. If performance is slow, this process significantly reduces productivity. Additionally, there is no strong set of tools to gather and analyze performance metrics. This is in contrast to Dafny, which can show how much work has been spent to verify a particular definition [26].

Parser Error Messages Parser error messages become unhelpful when pure functions are used with more involved definitions using let and ? as done in the standard library. For example, missing parenthesis in the definition of one function may result in an error in a completely different function that has no problems.

Chapter 4

Conclusion and Future Work

In this chapter, we briefly summarize our findings and suggest directions for future work.

4.1 Conclusion

Using `opaque` decreases the number of quantifier instantiations but not the execution time. However, this may be due to the relatively small scale of the examples tested. We generally observe that quantifier instantiations alone do not seem to correlate to execution time. As expected, turning off axiomatization and manually proving the proof obligations results in quantifier instantiations equivalent to proving an empty file and a reduction of execution time.

Assisting the verifier does not seem to have a clear effect on quantifier instantiations: while in some cases, the number of quantifier instantiations decreases, in other cases, it increases. Similar observations hold for the variability. In general, it appears that assisting the verifier hurts execution time, suggesting that we should minimize what is passed to the verifier for pure performance.

4.2 Future Work

As described in our discussion of the limitations of Gobra, the applicability of the standard library could be significantly improved by implementing support for generics, `closed` and more complete support of Go's scoping rules. Additionally, implementing packages for types like arrays, options, multisets, and packages for non-linear arithmetic may increase the usefulness of the standard library.

Furthermore, improved tooling for understanding the verifier's context and performance may allow for a more thorough evaluation of the standard library and keywords like `opaque` and `closed`. Porting Dafny's concept of work units may be a first step in this direction.

Finally, our analysis did not measure proof stability beyond performance. In particular, we did not measure how proof stability changes in terms of provability. Measuring this kind of stability and finding solid predictors to motivate concrete programming guides could be another work direction. One simple, straightforward way could be to measure provability across different Z3 versions.

Bibliography

[1] Dafny core library: Multiplication. https://github.com/dafny-lang/libraries/blob/master/src/NonlinearArithmetic/Mul.dfy. Accessed: 2023-10-12.

[2] Dafny homepage. https://dafny.org/. Accessed: 2023-10-12.

[3] Dafny standard libraries. https://github.com/dafny-lang/dafny/tree/master/Source/DafnyStandardLibraries. Accessed: 2024-03-07.

[4] Github – report quantifier instantiations. https://github.com/viperproject/silicon/pull/587. Accessed: 2024-03-25.

[5] Github – strange output when reporting quantifier instantiations using newer z3 versions. https://github.com/viperproject/silicon/issues/786. Accessed: 2024-03-25.

[6] Github – viperproject/gobra at da25624. https://github.com/viperproject/gobra/tree/da25624a260e5dbe86035ab9df42154e59c55567. Accessed: 2024-03-25.

[7] Github – viperproject/gobra-libs. https://github.com/viperproject/gobra-libs. Accessed: 2024-03-27.

[8] Github – viperproject/silicon at 0608ac9. https://github.com/viperproject/silicon/tree/0608ac922cf9d6b4b8dcfff6d31b7a66daa28e38. Accessed: 2024-03-25.

[9] Release z3-4.8.7. https://github.com/Z3Prover/z3/releases/tag/z3-4.8.7. Accessed: 2024-03-25.

[10] Verus: Verified rust for low-level systems code. https://github.com/verus-lang/verus. Accessed: 2023-10-16.

[11] Z3 guide: Quantifiers. https://microsoft.github.io/z3guide/docs/logic/Quantifiers/. Accessed: 2024-03-22.

[12] Linard Arquint, João Carlos Mendes Pereira, Peter Müller, Dionisios Spiliopoulos, and Felix Wolf. Gobra homepage. https://www.pm.inf.ethz.ch/research/gobra.html. Accessed: 2023-10-12.

[13] Linard Arquint, Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David Basin, and Peter Müller. Sound verification of security protocols: From design to interoperable implementations. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093, 2023.

[14] The Go Authors. Go homepage. https://go.dev/. Accessed: 2023-10-12.

[15] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 917–934, USA, 2017. USENIX Association.

[16] Secure Foundations Lab CMU. Verifying opentitan. https://github.com/secure-foundations/veri-titan. Accessed: 2023-10-12.

[17] Timon Egli. Translating pedagogical exercises to viper's go front-end, 2023. Bachelor's Thesis at Department of Computer Science, ETH Zürich.

[18] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[19] K.R.M. Leino and K. Leino. *Program Proofs*. MIT Press, 2023.

[20] ETH Zürich Department of Computer Science. Viper tutorial. http://viper.ethz.ch/tutorial/. Accessed: 2023-10-12.

[21] João Carlos Mendes Pereira, Peter Müller, Dionisios Spiliopoulos, and Felix Wolf. Verifiedscion homepage. https://www.pm.inf.ethz.ch/research/verifiedscion.html. Accessed: 2023-10-12.

[22] Viper Project. Gobra tutorial. https://github.com/viperproject/gobra/blob/master/docs/tutorial.md. Accessed: 2023-10-12.

[23] Microsoft Research. Ironclad. https://github.com/microsoft/Ironclad/tree/main. Accessed: 2023-10-12.

[24] Toccata. Why3 homepage. https://why3.lri.fr/. Accessed: 2023-10-12.

[25] Toccata. Why3 standard library. https://why3.lri.fr/stdlib/. Accessed: 2023-10-12.

[26] Aaron Tomb and Jean-Baptiste Tristan. Avoiding verification brittleness in dafny. https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/. Accessed: 2024-03-22.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

☐ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☒ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

☐ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

A Standard Library for Gobra

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Nezamabadi	Daniel

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zurich, 27.03.2024	Daniel Nezamabadi

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard