Extending the Specification Language of a Go Verifier with Algebraic Data Types

Bachelor's Thesis Project Description
Paul Dahlke

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

Start: 22nd Feburary 2021
End: 23rd August 2021

Introduction

Go is a systems programming language, developed to tackle problems with modern architectures like multiprocessor systems, networked systems, and massive computation clusters. While Go supports many modern features such as garbage collection or static typing [6], the Go compiler cannot prove the absence of implementation errors.

Gobra [2] is an automated modular verifier for Go programs, developed in the Programming Methodology Group at ETH Zurich. Gobra verifies memory safety, crash safety, data race freedom, and offers a specification language to express the intended behaviour of a program. These specifications are given in the form of method pre- and postconditions and loop invariants. A Go program is annotated with these specifications and additional proof annotations. Gobra verifies such an annotated program and then outputs whether the verification succeeded or not. If the verfication fails, Gobra reports back which part of the verification failed. Gobra encodes annotated Go programs into the Viper [5] intermediate verification language to perfrom verification.

Gobras specification language supports a variety of built-in mathematical types, namely sequences, sets, and multisets. Mathematical types are used to abstract over the state of data structures. For instance, the content of a linked list can be abstracted to a mathematical sequence. Even though Gobra support those mathematical types, custom data types cannot be defined.

To mitigate this issue, we add the support for *algebraic data types* to Gobra. Algebraic data types are supported by many programming languages and verification languages such as Haskell [3] and Isabell [1] to define new types. Consider the following definition of a tree type in Haskell:

```
data Tree = Leaf | Node Int Tree Tree
```

Listing 1: A simple Tree algebraic data type in Haskell

This defines a new type `Tree` with two *clauses*, `Leaf` and `Node`. Each of these clauses has a *constructor* to create values of the `Tree` type, namely `Leaf()` and `Node(i,l,r)` for some integer `i` and some trees `l,r`. Note that every value from the `Tree` type is either a `Leaf` or `Node` but nothing else.

Listing 2 shows an example for *deconstructors* in Haskell. Deconstructors are the counter part of constructors. For example, the deconstructor `getNodeValue` returns the integer value that is stored in a `Node`. Listing 2 also shows *pattern matching*. Pattern matching allows the programmer to match an arbitrary value of the `Tree` type against the constructors. In the example, we matched against the `Node` constructor to define the deconstructors.

Finally, listing 3 shows an example of *discriminators* in Haskell. Descriminators are used to determine the constructor of some algebraic data type value. In the example, the function `isNode`

```
getNodeValue t = case t of
    Node i _ _ -> Some(i)
    Leaf       -> None

getLeftTree  t = case t of
    Node _ l _ -> Some(l)
    Leaf       -> None

getRightTree t = case t of
    Node _ _ r  -> Some(r)
    Leaf        -> None
```

Listing 2: Deconstructors in Haskell

```
isNode t = case t of
    Node _ _ _ -> True
    Leaf       -> False

isLeaf t = case t of
    Node _ _ _ -> False
    Leaf        -> True
```

Listing 3: Clause check in Haskell

uses pattern matching and returns true if and only if the value `t` matches with `Node(_,_,_)`. `isLeaf` is defined analogously.

In this thesis, we aim to extend Gobra with algebraic data types (Goal 1). Furthermore, we aim at providing support for algebraic data types by not only extending the verification language but also the proof capabilities (Goals 2 and 3). In addition, we aim to evaluate the performance of the extensions (Goal 4) and lay out an argument for the soundness of our encoding of algebraic data types into Viper (Goal 5).

Core Goals

1. Design and implement algebraic data types in Gobra

The first goal is to enhance the specification language of Gobra. We will take the following steps:

1. Come up with an encoding of generic algebraic data types in Viper using Viper's *domains* feature. Domains are used to to introduce new types and consist of uninterpreted types, uninterpreted functions, and axioms on these functions. The goal is to find suitable functions and axioms, which model general algebraic data types, including constructors, deconstructors, and descriminators. Furthermore, these axioms have to be sufficiently complete, meaning that they do not contain contradictions, and reasonable complete, meaning that they suffice to prove the properties we are interested in. In Viper, forall quantifiers have to specified with *triggers*. Triggers are guiding the SMT solver to a quick solution, by restricting the instantiation of the quantifier. However, choosing bad triggers can lead to *matching loops* and therefore to a non terminating verification. Therefore, this goal includes choosing good triggers in our encoding of general algebraic data types.
2. Design a syntax for algebraic data types in Gobra. The Syntax should match the current specification language of Gobra.
3. Implement algebraic data types in Gobra. This includes extending Gobras parser, type checker, and the encoding to Viper.

2. Extend the support for algebraic data types

The second goal is to design and implement additional features to make working with algebraic data types more convenient. We will improve the proof support for algebraic data types. Concretly, we will add support for induction and case distinctions, i.e. pattern matching, on algebraic data types. To do so, we will add the following language constructs to Gobra:

- **Induction and termination** A common proof pattern for algebraic data types is induction. To allow sound induction on algebraic data types, it is necessary to define a well-founded order on algebraic data types. This goal includes pre-defining such well-founded orders. Similar efforts can be found in other proof tools like Dafny [4].
- **Pattern matching** A common language construct to improve the usability of algebraic data types is pattern matching. We aim to support pattern matching as an expression and as a statement. A pattern matching expression matches a value against the constructors and returns an expression for each case. A pattern matching statement works analogously.

3. Functions on algebraic data types

Tools which incorporate algebraic data types often offer build-in functions on them. Common functions are `depth`, `size` or `to_seq`. Even though these functions can be encoded by hand, we want to generate them automatically. We will pick a suitable set of functions that we can generate automatically, reducing redundancy when using Gobra. At this point in time, we aim to support the functions `depth`, `size`, `to_seq`, and `to_set`. For some algebraic data types, it does not make sense to generate all of these functions. One approach is to use a deriving system, similar to Haskell's, which allows to specify which functions should be generated.

4. Evaluation

The fourth goal is to evaluate the performance of our extensions. For performance, the challenge is to isolate the performance of our extensions. We will write example programs that only use our extensions and measure their verification time.

Furthermore, we will collect example programs that benefit from algebraic data types and measure their verification time. To isolate the overhead introduced by algebraic data types, we will subtract the verification time required without any specifications and proof annotations for algebraic data types.

Extension Goals

Compiling ghost code to Go code

In Gobra, so far, mathematical types have only been used for specification and proof purposes. Gobra programs can introduce auxiliary variables or fields that contain mathematical types, but they are classified as annotations. Therefore, these are not part of the verified Go program. In fact, Gobra provides command line options to move all annotations into comments and vice versa.

Mathematical types such as sets, sequences, and algebraic data types can be implemented in Go. That means that we can transform auxiliary code that uses mathematical types to standard Go code. This can be useful for assertion checking.

The aim of this extension goal is to add support for compiling ghost code to Go code. We restrict ourselves to sequences, sets, and algebraic data types. Other ghost constructs such as permissions or predicates are not considered for this goal.

Adding algebraic data type support for Viper

Viper has a plugin system to add new language constructs to Viper. The aim of this extension goal is to add support for algebraic data types as a plugin to Viper. We expect that we can reuse big parts of our implementation of the first and second core goal. However, we have to extend Viper's parser and type checker for this goal.

Term algebra with equational theory

As motivated in our introduction, algebraic data types are a language construct to enable users to define new custom types. However, not all types can be expressed as algebraic data types. In particular, the equality relation on algebraic data type values is fixed by the definition of the algebraic data type and cannot be extended with additional equalities. Consider the algebraic data type for simple mathematical terms shown in Listing 4. Adding the equality `Num 0 == Add 0 0` to our axioms is unsound because for an algebraic data type, all constructors are injective. If we want to support reasoning about such additional equalities, we require a different type construct than algebraic data types. *Term algebras* are such a language construct. The constructors of term algebras are not injective and as such they permit additonal equalities. Note that as a consequence, term algebras do not have deconstructors and discriminators. Together with a term algebra, we define an *equational theory*, a relation on term algebra values, that defines whether two values are equal or not. More concretely, for an equational theory E, two term algebra values x and y are equal if and only if they are structurally equal or (x, y) is contained in E.

```
data Exp = Num Int | Add Exp Exp
```

Listing 4: Exp type in Haskell

The aim of this extension goal is to add support for term algebras with an equational theory to Gobra. We will design a language construct to define an equational theory such that Gobra can use them for verification. Fully automating an equational theory is hard for SMT-based verification approaches. The challenge is to avoid matching loops in the encoding that lead to a non-terminating verification. The aim of this extension goal is to add basic support for term algebras with equtional theory. We prioritize termination over automation, i.e. the user might have to explicitly annotate used equalities.

Apply developed techniques to other Go datatypes

In Go, algebraic data types can be implemented through a combination of structs and interfaces. We decided to add a separate construct because we believe this leads to a better language design and introduces less overhead. That said, techniques developed for algebraic data types, in particular induction patterns, can also be applied to structs and interfaces. For this extension goal, we add support for structural induction over data types other than algebraic data types.

References

[1] Julian Biendarra et al. *Defining (Co)datatypes and Primitively(Co)recursive Functions in Isabelle/HOL*. URL: `https://isabelle.in.tum.de/doc/datatypes.pdf`. accessed: 09.03.2021.

[2] *Gobra*. URL: `https://www.pm.inf.ethz.ch/research/gobra.html`. accessed: 28.02.2021.

[3] Cordelia Hall and John O'Donnell. "Introduction to Haskell". In: *Discrete Mathematics Using a Computer*. Springer, 2000, pp. 26–29.

[4] K. Rustan M. Leino. *Dafny Power User: Automatic Induction*. URL: `http://leino.science/papers/krml269.html`. accessed: 28.02.2021.

[5] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2016, pp. 41–62.

[6] Rob Pike. *Go at Google: Language Design in the Service of Software Engineering*. URL: `https://talks.golang.org/2012/splash.article`. accessed: 01.03.2021.