Parser for Go Programs and Specification

Bachelor Thesis Project Description

Nico Berling

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

Start: 15th September 2021
End: 15th March 2022

1 Introduction

Gobra is a verification tool for the systems programming language *Go*. Gobra supports regular Go memory manipulation as well as Go's standout features, such as goroutines, channels, and wait groups [1]. This is accomplished by encoding Gobra programs, i.e. Go programs with added specifications, to Viper programs, and then verifying these programs with an existing Viper backend.

The first step of this process consists of parsing the program and creating an abstract syntax tree. The current implementation uses a parser written purely in Scala. This approach suffers under multiple issues: First of all, the performance of this parser is unsatisfactory, because it is based on parser combinators. Parser combinators are useful to quickly prototype and test grammars, but can suffer from bad performance. Secondly, the currently implemented grammar does not handle all Go code correctly, and requires some workarounds for ambiguous rules.

This thesis aims to improve upon this by writing a new parser that can parse more programs and do so more efficiently and more user friendly than the current one.

2 Context

The parser combinator library that is currently used by the project is Kiama [2]. It is part of a research project attempting to embed grammars and other formalisms directly into programming languages, in this case Scala. Parser rules are written as applications of first-order Scala functions, each representing one standard operator in a grammar specification, such as concatenation, repetition, alternatives and so on.

This gives the resulting code a shape similar to the formal specification of the language. For example, the code for a simple production rule $\textit{summand} \rightarrow \text{summand}\{+\text{summand}| - \text{summand}\}$ looks like `summand ~ rep("+"~summand | "-"~summand)`, where `~` is the function for concatenation, `|` for alternative and `rep` for repetition.

This makes the parser very easy to understand and modify, but comes at the cost of performance: Because the parser rules are not preprocessed in any way, the parser is essentially constructed anew for each input. Clever caching can soften this impact, but it is still not comparable to optimized parsers generated by other methods. [3]

This issue impacts the performance of Gobra significantly. Parsing simple files can take several seconds, sometimes longer than the verification process itself. Experiments have also shown that the parser scales superlinearly with the size of the resulting abstract syntax tree. On the other hand, because the parser is incomplete, programs that conform to the Go specifications have to be rewritten to avoid Gobras inability to parse some constructs. The following sections expand on these problems and propose potential solutions.

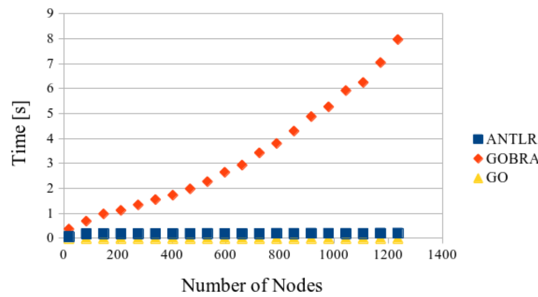

Figure 1: Performance comparison of parser alternatives on generated test files

```
1  package pkg
2  type sliceStruct struct {
3    f []int
4  }
5  func test(s sliceStruct,j int) int {
6    return s.f[j] // has to be replaced by "(s.f)[j]"
7  }
```

Figure 2: Indexing a selector expression in Gobra

2.1 Currently Known Problems

2.1.1 Exponential performance

As mentioned in the introduction, parser combinators exhibit exponential performance in the worst case. Figure 1 shows the performance of Gobra's parser measured on a series of test files constructed to create ASTs of increasing size. The programs consist of a variable amount of functions, each consisting of a single assignment and a return statement.

These tests are not fully representative of real life performance, but already give an indication of the problem at hand: The current grammar and parser library seem to be reliant on backtracking, even though it should not be necessary for a language that allows backtracking-free grammars. The tests also show that ANTLR, a parser generator framework, can parse pure Go programs significantly faster than Gobra's current parser.

2.1.2 Indexing selector expressions

In the Go language, struct fields are accessed by selector expressions of the form `x.f`. According to the specificaton, if the accessed field is indexable, for example if it is an array, the index can be written directly after the selector statement, e.g. `x.f[i]`. In Gobra, however, you must surround the statement by parentheses, e.g. `(x.f)[i]`, otherwise the parser will fail irrecoverably, see Listing 2.

2.1.3 Struct Literals in Conditional Expressions

In Go, the conditional expressions of control statements are not surrounded by parentheses. At the same time, Go supports composite literals of the form `Point{9,8}`, where `Point` is the name of a struct type. This leads to a syntactic ambiguity: code of the form `if a==b { f(x)} { f(y) }` is either an if statement with the condition `a==b`, the body `f(x)` and an unrelated block with a single expression `f(y)`, or it has the condition `a==b{f(x)}`, where b is the name of a struct type, and the body of the if statement is `f(y)`.

Go and Gobra interpret these ambiguities differently. While Go never accepts struct literals directly inside conditional expressions, Gobra will interpret curly brackets inside conditions as a part of struct literals whenever feasible, see Listing 3. This is hard to fix in the current Gobra parser, because it is mostly context insensitive.

```
1  package pkg

2  type Simple struct {
3    x int
4  }

5  func fooo(s Simple) {
6    if s == Simple{0} {      // This condition must be enclosed in parentheses to be
7      point = Point{0,0}     // accepted by the Go parser, but the Gobra parser
8    }                        // accepts it as is
9    x := 4
10   if point.y == x {        // This condition must be enclosed in parentheses to be
11   }                        // accepted by the current Gobra parser but the
12                            // Go parser accepts it as is
13 }
```

Figure 3: Diverging behaviour on struct literals inside control clauses

2.1.4 Akward Syntax for First-Class Predicates

Gobra supports first-class predicates, meaning that predicates are values themselves and can, for example, be passed as arguments. They can be constructed by supplying some or all arguments to a predicate and leaving the rest blank as wildcards. The intended syntax for such a predicate constructor is as follows: `P1{d1, ..., dn}`.

If we have a predicate `pred` that takes three integer arguments for example, we could construct a new predicate that only takes one argument by supplying the first and the last argument `pred{5,_,9}`. This introduces a similar problem as the conditional expressions: if we had supplied all three arguments, the constructor could just as well be a struct literal. Therefore, Gobra currently uses the alternate syntax `P1!<d1,...,dn!>`.

Resolving this requires type information, and therefore parts beyond the parser would have to be modified to support this proper syntax.

2.2 Alternatives

As opposed to parser combinators, where the entire grammar is essentially contained in the code itself, parser generators take as input a formal grammar of the language, and generate the source code for a parser for this grammar. The formal specification is usually written in some framework-specific language that is quite similar to EBNF. The resulting source code contains optimized functions that can then be integrated into the larger project and compiled with the rest of the project.

This approach is faster than parser combinators, because the construction of an optimized parser is done in advance. This means that the performance is dependet entirely on the generated parser itself, and not on the construction of the parser. One such framework, ANTLR4 [4], seems especially well suited, since it can generate the parser code in Java and has existing grammars for Go. This generated parser can be more easily integrated in the existing pipeline than alternative parser generators such as YACC which generates C code.

An alternative to parser generators are handwritten recursive descent parsers. In particular, we could extend the exiting Go parser so that it can parse Gobra's additional annotations. This involves significant effort in the later step of integrating the parser into the existing pipeline, as the Go AST generated by the Go parser has to be converted to the existing format of the Scala AST already used in the rest of Gobra.

3 Goals

3.1 Core Goals

(1) **Performance Assessment.** The first goal is to assess and profile the performance of both any relevant alternatives and the current parser, to better understand its deficiencies. We do this by running the parsers on real world sources from Github, for example the SCION project, and programs we generate for evaluation purposes. Furthermore, we evaluate the quality of error messages and estimate the work required to integrate them into Gobra. In

particular, we evaluate the effort neccessary to extend Gobra with new language constructs later and the effort neccessary to adapt the parser to changes in the Go language.

(2) **Implementation.** We will then implement the new parser in the chosen framework. For most of the parsing frameworks, grammars for Go itself already exist, these must be extended to support Gobra annotations. The resulting parser has to be able to parse all Go code, including packages, and all Gobra annotations. In particular, the parser should fix the known problems listed in 2.1. The new parser will be well documented and maintainable, so that it can be extended with new language constructs later.

(3) **Integration.** We integrate this new parser into the existing Gobra pipeline as seamlessly as possible. This means the parser must construct an abstract syntax tree that is of the same format as the one that the current parser provides. Depending on the framework, this will include a translation from the internal AST of the framework to Gobras current AST. The integration must allow for future modifications.

(4) **Better Errors.** The new parser should issue more informative error messages for parse failure. This is generally limited by the parser framework, but can be improved upon. The aim of this goal is to guarantee that the messages contain enough information about parse errors, both in the program code and the specifications, to allow for easy troubleshooting.

(5) **Evaluation.** We will test the new implementation against a large set of both real world sources and constructed test programs. Some parts of this evaluation will focus on the parser itself, and others will also include the entire Gobra pipeline. To do this, the existing Gobra benchmarks can be used. Furthermore, we do a textual evaluation of the error messages.

3.2 Extension Goals

(1) **IDE support for code navigation.** In order to provide functions such as “go to definition”, the parser needs to provide certain information quickly. This goal is to provide this information to the Gobra IDE to allow for easy code navigation.

(2) **IDE support for partial programs.** To give better feedback during development, the parser should be able to parse incomplete programs as far as possible and already report on problems, such as type errors, as early as possible. This goal is to add such support for partial progams.

(3) **Fix context-dependent issues.** Some parsing issues are not strictly limited to the parser itself, but instead involve the type checker. One such example is the akward syntax mentioned in 2.1.4. In this extension goal, we adapt the type checker to better handle these ambiguities.

References

[1] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, “Gobra: Modular specification and verification of go programs,” A. Silva and K. R. M. Leino, Eds. Springer International Publishing, 2021, pp. 367–379.

[2] “inkytonik/kiama: A scala library for language processing.” [Online]. Available: https://github.com/inkytonik/kiama

[3] M. Odersky, *Programming in Scala*, fourth edition ed., L. Spoon and B. Venners, Eds. Artima, 2019.

[4] T. T. J. Parr, *The definitive ANTLR 4 reference*, 2nd ed. The Pragmatic Bookself, 2012.