Type inference for Go

Project in Computer Science - EDAN70

Emin Gigovic
D11, Lund Institute of Technology Sweden
dat11egi@student.lu.se

Philip Malmros
D11, Lund Institute of Technology Sweden
dat11pma@student.lu.se

Abstract

Automatic deduction of a data type from a given expression is in programming referred to as type inference. How advanced the type inference varies between languages, it can go from being a ”nice to have” feature, to being a core part of the language. A subset of the statically typed language Go was implemented using JastAdd, with focus on type inference. By automated tests and comparison of how well type inference works in other languages, the subset was evaluated in a primarily qualitative manner.

Keywords Type inference, Go, JastAdd, Reference Attribute Grammars

1. Introduction

The ability to automatically interpret the type of an expression in a programming language at compile time is the definition of type inference, which in general is characteristic for functional programming languages like Haskell[1]. The advantage with type inference in a programming language is primarily the freedom to omit types, which makes several programming tasks easier. Two characteristic type terms used in this context are: dynamic types and static types. Static typing means that the type of a variable is known at compile time and dynamic typing means that the variable type is interpreted at runtime.

1.1 Problem Description

A type system consists of rules that assigns types to variables depending on their computed value. Type inference in Go can only be used for initialization of variables and constants. The purpose of this project is to implement a subset of Go using JastAdd with focus on type inference and doing a qualitative evaluation of the implementation by various measurements, comparisons towards other programming languages and presenting useful examples of inference.

2. Background

2.1 Go

Go, or often called golang, is a statically typed language with garbage collection, various safety features and the possibility to use concurrent programming. The syntax is highly influenced by C and C++, and the goal of the developers was to create a language which is concise, simple and safe. Some of Go's touted features include: fast compilation time, concise declarations and initializations of variables through type inference, built in concurrency modules and remotely managing packages[2]. The team behind Go aimed to improve the working environment for their designers and programmers by making the process at the software development more productive by the use of Go to eliminate slowness and clumsiness. Common critique of the language include lack of particular features which makes it hard to use the language in large-scale software development. Some examples of ”missing” features are: lack of extensibility like operator overloading, uncontrolled dependencies, limitations of use in system programming due to the functionality of the used garbage collection and the absence of a Hindley-Milner type system in the implementation of the type inference system.

2.2 Type inference

Explicitly writing what type variables have at their definition is common in programming. However, using type inference the compiler can often handle this part on its own. To what extent the compiler can do this varies from language to language, but the principle behind type inference is to iteratively derive some valid type for the language from an expression containing one or more values. Some examples of what can be inferred by a compiler in this way are values for: variables, parameters and return statements[3].

The actual point of having the compiler support something like type inference can be broken down into two main parts. First of all, when used properly it can make code much more readable, for instance by reducing this C++ code:

```
vector<int> v;
vector<int>::iterator itr = v.iterator();
```

to this

```
vector<int> v;
auto itr = v.iterator();
```

While the gain might look insignificant here, if the type had been more complex but still intuitive with properly named variables, the value of type inference starts to become apparent as in many cases it would let us reduce redundant information in our code. This level of type inference is on the lower end of what is possible however, some languages use type inference for other features in addition to readability. Haskell is one such language, in which it is possible to write:

```
succ x = x + 1
```

This gives a function *succ* that will take x (regardless of what type x has), add one to it and then return the result. Explicit type specifiers could still be used for more advanced functions, so the compiler has an easier time understanding what the code is actually supposed to do, and is less likely to do any mistakes[4].

2.3 Type inference in Go

As mentioned above, type inference can be implemented to varying extent, and in Go's case the implementation is rather bare bones.

According to Go's developers they were aiming to reduce the clutter often found in statically typed languages when they designed their type system. One of the major reasons for this seems to have been that they felt that many programmers were finding the type systems found in languages like Java or C++ too cumbersome, and that they preferred the approach in dynamically typed languages. So when designing Go they borrowed some of the ideas from these languages. One of these ideas was to use simple type inference for variables, giving the feel of writing dynamically typed code, while still using the benefits of static typing[5, 6]. As mentioned in the section above, type inference can also cover things like parameters and return values, but this is absent in Go.

In practice the type inference in Go can be triggered by either simply leaving out the type information when declaring a new variable or constant, or by using the ":=" notation, which is a shorthand for declaring a new variable (there is no shorthand for constants) with an inferred type. In Go the following three statements are equivalent:

```
var a int = 10
var a = 10
a := 10
```

Go's type inference is a bit half-done regarding how it handles inferring values containing identifiers. Essentially the compiler will not allow type casting on values retrieved from identifiers, so to give a few examples:
This code runs without problems, and *a* becomes a float64 value:

```
a := 1 + 1.1
```

No problems below either:

```
a := 1.1
b := 1 + a
```

However, this code will give an error that the value used from *a* has been truncated to an Integer, instead of casting *a* to a float64, giving *b* the type float64 in the process:

```
a := 1
b := a + 1.1
```

There is a similar problem here:

```
a := 1
b := 1.1
c := a + b
```

Instead of casting *a* to a float64, the compiler will find that the types don't match and gives an error.

2.4 JastAdd

JastAdd[7] is a compilation system that supports Reference Attribute Grammars (RAGs) used for generating language-based tools, such as compilers. Explicit definitions of graph properties in a program is used because of the existing support for reference-valued attributes, which is an important feature in the system. Properties of abstract syntax nodes can be programmed declaratively, attributes have values defined by equations and reference values points to other nodes in the abstract syntax (AST). Attributes can be defined either as synthesized or inherited depending on if the information should be propagated upwards or downwards in the AST. The order of declaration of attributes and equations have no influence of their meaning which allows the programmer to organize the code into modules for reuse and composition. Data structures are embedded in the AST, which results in an object-orientation of the program. Java is integrated in JastAdd and the resulting model is implemented using Java classes to form a method API for the attributes. The advantage of declarative programming is the ability to easier add extensions to an existing language.

3. Implementation

The Go subset is implemented with the help of JastAdd, the parser generator Beaver and the scanner generator JFlex. We used JastAdd to implement attributes for the abstract syntax tree (AST), semantic analysis, error analysis and the interpreter. The semantic analysis consists of type analysis and name analysis. The purpose of name analysis was to define attributes for checking multiple declarations or undeclared variables and functions. Type analysis has the functionality of checking if the type assigned to a variable is valid, if parameter and return type is correct or exists if needed. Supported types in our subset are int, bool, float64 and void. Int and float64 are used for basic arithmetic operations while void is used for declaring functions, stating that the return value can be omitted.

```
func main() {
  var a int = 5;
  var b = a + 2;
  c := a + b;
}
```

Compile-time errors are implemented via collection attributes. Every contribution is automatically collected by the attribute evaluator, traversing the AST and adding each contribution to the collection. Only relevant errors are reported which implies that errors caused by other errors are not reported. Example of relevant errors are mismatched types, undeclared variables or functions, missing or incorrect type on the return statement, mismatch on the parameter values and possible dead code in a block.

To extend what would be possible to use in code examples, and since they can use type inference, our subset also includes global and constant variables.

```
var a = 5;
var a2 = true + 3;
var c = 5.9;
var f float64 = 2;

func main() {
  var b int = a;
  var d bool = a + c; // Will give an error
  var g int = f; // Will give an error
}
```

```
func main() {
  const a int = 2;
  a = 1; // Error
  const b float64 = 3.14;
  b = 5; // Error
  const c bool = false;
  c = true; // Error
  const d = 5;
  d = 2; // Error
  const e int = b + c;
}
```

Regular control structures like *if* and *for*-statements are supported by our subset. A *while*-statement in Go is defined as a *for*-loop consisting of only one condition, and is thus supported as well. A curious aspect regarding *if*-statements in Go is that, like in a *for*-

statement, you can define a statement that is only executed once at the start of the *if*, the result of which is only in scope for the rest of the if-statement.

```
func main() {
  if i := 1; i > 10 {
  }
}
```

```
func main() {
  var i int = 1;
  for i2 := 1; i2 < 10; i2 = i2 + 1; {
   i = i + 1;
  }
}
```

As we mainly focused on type inference, we never got around to implementing the package system that Go utilizes to make use of functionality in other files. Since the Println function in Go uses a package, this meant that we had to make our own (albeit) simple predefined printing function that is completely separate from Go. Our Print can only handle one parameter, a variable of some sort, and will simply print the value of this variable.

An interpreter was implemented with the help of various attributes, and executes a program by traversing the corresponding AST and outputs values or errors to the user via a print method.

4. Evaluation

4.1 Functionality

To the extent of our subset, we have as far as we can tell almost equivalent functionality to Go, including how the type inference system works. For instance, while Go supports type casting for inference of constant values, it does not support type casting for values that comes from used identifiers. Our subset however, is unable to do any type casting whatsoever, as we had missed that there was a difference between the two cases for actual Go until very late in the project. Constants in our subset also has the unintended quirk of being able to be given values that does not necessarily come from another constant value. Apart from these two cases we have not found anything noteworthy that differs between out subset and an equivalent slice of the actual language.

Originally it was intended that we were to try and make improvements on how Go handles some things concerning type inference. The two first things to start with was trying to fix the odd and inconsistent approach to inferring values from identifiers used in Go, as well as attempt to add inference for parameters and/or return values. Unfortunately we had issues solving some other parts of the implementation, and so never really got the time to make a serious effort to make these improvements.

4.2 Handling errors

The treatment of errors works in such way that only relevant errors are reported, which means that errors caused by other errors are not reported. Relevant errors are undeclared variables or functions, type conflicts, wrong or missing type on the return-statement, mismatch on the parameter-values or possible dead code in a block. Below are different code samples which shows how errors are handled in different situations. In the first code example there are mismatched types, the second one has the wrong return type the third example has a incorrectly declared main function.

```
func main() {
  var a int = 5;
  var b int;
  var c float64 = 10.5;
  b = a + 2;
  a = b + c;
}
```

```
Error at line 6: Invalid operation: b + c (mismatched
     types: int and float)
```

```
func main() {
  var a = 1;
  return a;
}
```

```
Error at line 4: Trying to return int in void function
```

```
func main() int {
}
```

```
Error at line 1: Main function should be void
```

The next code example illustrates how the error handling and type conflict implementation differs from our subset compared to the original Go compiler.

```
func main(){
   age1 := 20
   age2 := 20.5
   age3 := age1 + age2
   age4 := 20 + 20.5
}
```

Running this code in our compiler yields the error messages below as our subset is unable to use any casts at all.

```
Error at line 4: Invalid operation: age1 + age2
     (mismatched types: int and float)
Error at line 5: Invalid operation: 20 + 20.5
     (mismatched types: int and float)
```

The difference compared to the original Go compiler is that the error at line 5 is not reported since it is a valid statement according to their implementation.

4.3 Lines of Code

The implementation of our subset was evaluated with respect to the number of lines of code with the evaluation tool Cloc[8]. Cloc counts the amount of code including blank lines, comment lines and physical lines of several languages like JastAdd, Java, Beaver, Flex and Go. Table 1 shows various measurements for the implementation of our subset and table 2 shows the measurements of the labs in the compiler course EDAN65 for comparison, since our subset is based on the code from the labs and covers a subset of similar size.

Language	Files	Blank	Comment	Code
JastAdd	8	170	21	765
Java	10	98	129	440
Beaver	1	54	1	200
Flex	1	11	8	61
Abstract grammar	1	11	0	48
Sum	21	344	159	1514

Table 1. Lines of code for our subset

Language	Files	Blank	Comment	Code
JastAdd	12	196	62	936
Java	7	62	86	380
Beaver	1	32	1	105
Flex	1	11	8	51
Abstract grammar	1	9	0	34
Sum	22	310	157	1506

Table 2. Lines of code for the labs in the compiler course EDAN65

One thing to note is that while the total sizes might not be that different, it has to be taken into account that the code for the Go subset is extensively refactored compared to the code from EDAN65.

5. Related Work

As this project only implemented a subset of a language with such a narrow focus, especially considering how relatively simple the type inference is in Go, we had difficulties finding any other projects that had done something similar. In a broader sense however, our project can be related to other smaller example language implementations that make use of attribute grammars, like SCHADOW[9].

6. Future Work

After ironing out some known problems with our subset, future improvements could include extending it with more areas from Go. Every Go program is made up of packages that are used in the code with import paths, and adding correct package handling is a possible extension to our subset. Another important addition would be things like arrays and structs, so it would be possible to represent some proper data.

When the subset would be a bit more fleshed out it would probably be a good time to try to do some additions to the actual type inference system, so that casts of values from identifiers are handled better. Eventually support could be added for parameter and return-value inference.

While the possibility to output the Abstract Syntax Tree(AST) exists in our subset, for large files it is a bit hard to get a clear overview of the AST which makes it more difficult to interpret the output. Improving the presentation of this information is certainly something that could be done if development would continue.

7. Conclusion

The purpose of this project was to implement a subset of Go, using JastAdd, including type inference and evaluate it in a qualitative way by examining how advanced the type inference in Go actually is and comparing it to other languages by useful inference examples. How advanced the support for type inference is varies from language to language, in which we can now state that the type inference in Go is not very advanced, rather it is quite bare bone compared to Haskell and Scala, which both are use more high-level type inference. Limitations that were found after implementing and evaluating the type inference in Go was that only the right hand side of an expression is evaluated and it is only possible to use type inference at initialization of variables.

The implementation of the subset was based on an earlier implementation of a simple language called SimpliC and from the evaluation table of lines of code for respective part, we can see above that the total sum of code does not differ that much, where the amount of JastAdd code is lesser and amount of Beaver code has increased a bit. The reason for this is that a majority of the code was reused but refactored to a higher coding standard while new functionality was added, which made the amount of parser (Beaver) code increase.

A future project based on the existing code from this project could be to increase the subset of Go by covering several other features in Go, adding more type inference supprt as well as improving the output printout considering the available AST option.

Acknowledgments

We would like to thank Niklas Fors for supervising this project.

References

[1] https://www.haskell.org/, .

[2] Rob Pike. Go at google: Language design in the service of software engineering: Keynote, splash, 2012. URL http://talks.golang.org/2012/splash.article.

[3] Luca Cardelli. Basic polymorphic typechecking. *Sci. Comput. Program.*, 8(2):147–172, 1987. URL http://dx.doi.org/10.1016/0167-6423(87)90019-0.

[4] https://www.haskell.org/onlinereport/decls.html#type-signatures, .

[5] Rob Pike. Less is exponentially more, 2012. URL http://commandcenter.blogspot.de/2012/06/less-is-exponentially-more.html.

[6] https://golang.org/doc/faq.

[7] Görel Hedin. An introductory tutorial on jastadd attribute grammars. *Postproceedings of GTTSE*, 2010. URL http://fileadmin.cs.lth.se/sde/publications/papers/2009-Hedin-GTTSE-preprint-tutorial.pdf.

[8] A. Danial. Cloc— count lines of code, 2009. URL http://cloc.sourceforge.net/.

[9] S. Doaitse Swierstra Arie Middelkoop, Atze Dijkstra. Iterative type inference with attribute grammars. *GPCE'10, October 2010*. URL http://dl.acm.org/citation.cfm?doid=1868294.1868302.