University of Texas

Turing Scholars honors thesis

Parametric Polymorphism in the Go Programming Language

Matthew Allen

Dr. Jan S. Rellermeyer – Faculty adviser

May 18, 2016

Abstract

An extension to the Go language was developed that introduces parametric polymorphism in the form of generic functions. The changes to the language and the compiler needed to implement the type system extensions are discussed, and alternative implementation strategies are described. The resulting implementation of generic functions is backwards compatible with the existing Go standard and is consistent with the design goals of the language. The overhead of the current prototype implementation is assessed based on two commonly used programming patterns.

1 Introduction

Statically checked, typed-safe languages offer the programmer the ability to reason about the kinds of data that a program operates on and ensure that only valid data is used to perform calculations. When using a language with a static type system, there are guarantees of correctness that are offered, and type errors can be detected at compile time rather than at run time [10]. As with any form of static analysis, compile time type checking cannot account for all runtime behavior of the program, and so it must be conservative in rejecting potentially valid programs to ensure that type errors cannot occur. If a language implements a limited static type system without support for high level abstraction, the restrictions of the type system can prevent the programmer from writing abstract or general code. In this case, the type system inhibits programmers who use higher level abstraction, rather than assisting them. By adding more power to the type system, the language can become more expressive and the type system can become less of a burden on the programmer. Parametric polymorphism is a common feature of static type systems that allows functions or data structures to be written generically, so that they can operate on values of many types while still maintaining static type safety [1]. Generic functions and data structures have type parameters which are substituted for specific types when the function or data structure is used. These type parameters may be bounded, so that only a certain class of types can be used for a type parameter, which allows the generic construct to use some of the common features of its parameterized types.

The generic function implementation described allows for more expressive implementation of common programming practices. One example of a use case for generic functions supported by this implementation is writing functions that deal with collections without interacting with the element

type. Using the generic function system, a sort function could be written as follows:

```
func sort<T Comparable>(in []T) []T { ... }
```

Here the `sort` function operates on lists of comparable elements without relying on any specific element type.

2 Motivation

The Go programming language features a static type system, but the semantics of the type system are limited [7]. Without a way to use parametric polymorphism, it is impossible to write statically type checked functions that can be reused for multiple similar types. This leads programmers to either duplicate code many times to create separate, specialized versions of otherwise identical functions, or to bypass the static type system using casting and reflection [5]. This paper outlines an extension to the Go language that includes generic functions that are statically checked and transformed into standard Go code at compile time.

The design goals for the Go language include clear semantics and ease of use, as well as runtime performance and concurrency support [11]. The proposed extension to the language introduces additional complexity to the type system, but given the prevalence of parametric polymorphism in statically typed languages and the relative simplicity of this generic system, the type system is still understandable enough to satisfy the ease of use goal. The generic function implementation is also backwards compatible with existing Go syntax and semantics, so it is not necessary to update existing code to use the generic system. Runtime performance is an important consideration for the language extension, and is discussed in Section 7. The goal of this initial implementation is to serve as a proof of concept for the proposed type system extension, so performance was a secondary concern for the first version.

Since Go is a compiled language, changes to the language specification must be accomplished by changing the compiler(s) that implement the language. The new language constructs necessary to implement generic functions require changes to the grammar that defines valid Go programs. Because of this, the parser for the language must be updated, and new node types must be added to the abstract syntax tree (AST) of the language. These changes to the parser are outlined in section 4. Once the compiler can parse the new language constructs, the type checking system must be

updated with the new type rules. These changes to the type checking system in the compiler are detailed in section 5. Section 6 explains the changes needed to generate generic code correctly, as well as the trade offs for different code generation strategies.

3 Compiler Architecture

There are several compiler implementations for the Go language, each with different advantages and drawbacks. The main compiler is referred to as the Go Compiler, or GC, and is the original version of the compiler. Originally written in C, GC was made self-hosting in Go version 1.5, and it is now written almost entirely in Go [6]. Unfortunately, the process for transitioning GC from C to Go was mostly automated, and as a result the codebase has not truly made use of the features or advantages that Go has over C. One consequence of this is that GC uses hand-crafted parsing and type checking code derived from the C version despite the existence of parsing and type checking packages in the Go standard library. The code for performing these tasks is relatively tightly coupled with other parts of the system. For this reason, adding new features to the GC system is difficult, and a significant amount of work would have to be dedicated to working around the complexities of the C-like software design.

Another major compiler implementation is GCCGo, which is a Go frontend for the GNU Compiler Collection (GCC) [3]. GCCGo is written in C++ in the style of the GCC codebase, and it does not make use of the parsing and type checking libraries from the Go standard library. As a GCC frontend, the compiler inherits much of the complexity that is required of such a flexible compiler, and modifications to it must account for this overhead.

A less commonly used community compiler, LLGO, was chosen to be modified. LLGO is a frontend for the LLVM compiler system [9] that is written in Go, and it uses the standard library parsing and type checking packages. Because it uses these standard components, it was the easiest system to add modifications to and extend. Since the goal of this project was to provide a proof of concept for a generic type system in Go, a choice was made to sacrifice widespread usage in favor of ease of development and prototyping speed. LLGO implements all of the features of the Go specification, and uses the same Go runtime as GCCGo.

The LLGO compiler consists of the implementation of the Go runtime and a frontend compiler that emits LLVM intermediate representation (IR). The main program emulates the command line API of GC and GCCGo,

providing support for resolving dependencies, building packages, and linking programs. When a program is processed by the compiler, it first invokes the parser on the input files to generate a canonical AST for the program. Next, the type checker is invoked on the AST, which generates type information for each expression and definition and ensures that all of the rules of the type system are followed. Once the type information has been generated, the AST is transformed into a single static assignment (SSA) [2] form, which makes the AST more compatible with the SSA form of the LLVM IR. At this stage the LLVM specific translation occurs, generating IR from the SSA form of the program that can be fed into the LLVM system. Finally, the LLVM system outputs an executable, binary version of the input program. This architecture is designed to support a high level of modularity and separation of the various parts of the compilation process. The modularity of the system makes it much easier to implement new features in discrete parts, and provide ways to test the individual parts of the system in isolation.

4 Parsing

The first change that was required to implement generic functions was to extend the parser to support the new language constructs. The Go standard library contains a module, `go/parser`, that implements a parser for the Go language grammar. The grammar was designed to avoid ambiguity, and the parser is a recursive descent parser with single element lookahead and no backtracking. For reasons discussed below, a naive grammar for generic functions would introduce ambiguity, and would require a backtracking parser to implement. To avoid an extensive redesign of the existing parser, the extensions to the language were constrained to keep the grammar within the requirements of a non-backtracking parser. The consequence of this is that new rules must not introduce ambiguity between productions, and compromises were made to the syntax of the extensions to ensure that minimal changes to existing parsing code were required. The new grammar is backwards compatible with the existing grammar. The complete modified grammar can be found in Appendix A.

The syntax for generic function definitions and for specifying type parameters at call sites is similar to the generic syntax in Java and C#. Angled brackets ('<' and '>') were chosen as delimiters for the type parameter sections in function signatures and calls. The other common delimiters ('(', '[' and '{') are all part of existing language constructs that are valid after identifiers (function calls, index expressions, and composite literals, respec-

tively) so they could not be used without introducing ambiguity or making more extensive changes to the grammar. Angled brackets also have the advantage of producing less confusion for the programmer, as they are not used as delimiters anywhere else in the grammar. The function type signature, which includes the arguments and return type of a function, was extended to include an optional section for type parameters and their type bounds. This allows type parameters to be included in both function definitions and function literals, as well as function types for fields and parameters.

Unfortunately, the ‘<’ character cannot be unambiguously parsed as either the start of a list of type arguments or an operator when it occurs after an identifier. For this reason, the type arguments to a function call are supplied as part of the argument list, after the opening parenthesis of the call expression. This syntax is inelegant, but it allows for unambiguous parsing with minimal grammar rewriting, and the type inference features of the language mean that in many cases the entire type parameter section of the call can be omitted and inferred based on the argument types.

One other issue that had to be addressed to allow for these grammar extensions is introduced by the lexer, which translates the source file into a list of tokens for easier parsing. Since ‘>>’ is a token used for bit-shifting, the parser must be able to accept ‘>>’ or ‘>’ when consuming closing brackets in nested type parameter sections. This is accomplished by tracking whether or not the parser is inside a nested type parameter list, and allowing ‘>>’ to close the inner and outer parameter lists. Using this system, there is no risk of the right shift operator being mistaken for a part of a parameter list, as it is not valid in type signatures, and any combination of single or double angle brackets can be parsed correctly.

5 Type Checking

Once the new language constructs were correctly parsed into AST nodes, the type checker was updated to respect the new typing rules. The type checking module of the standard library, `go/types`, implements a type checker that associates each expression and definition in the AST with a type and verifies that all of the type rules are obeyed. The type checker recursively processes each node in the AST, building up types for complex expressions from their constituents. The system also infers types if they are omitted in the short form of variable assignment.

In generic function definitions, an extra parameter list is included that contains named type parameters and their type bounds:

```
func foo<A Bound>(x A) A
```

In this example, `A` is the type parameter, `Bound` is the type bound of `A`, and `foo` has type $\texttt{A} \rightarrow \texttt{A}$. Currently, each type parameter has exactly one type bound, which describes the operations that the type parameter must support. Within the body of the function, the type parameters are assumed to implement the interface of the type bound, but are otherwise unknown types. This means that values of type `A` support the operations defined on interface `Bound`, but `A` and `Bound` are not considered to be identical types. As a consequence, complex types such as slice[1] types `[]A` and `[]Bound` are not assignable to each other.

At the call sites of a generic function, each type parameter is instantiated with a specific concrete type. Instances of the type parameter are substituted with the specific type argument for that call site, and the function call is type checked using the usual rules. If `S` is a type that implements `Bound` and `val` has type `S`, then the call expression

```
foo(<S>, val)
```

uses `S` as the specific type argument for parameter `A`. In this case, the type of the complete call expression is `S`. In functions where the type parameters are used in the formal parameters of the function signature, their types can be inferred from the types of the arguments to the function. In this case, the function call could be written as `foo(val)` without changing the meaning.

5.1 Similar Type Systems

This implementation of generic functions is similar to the generic implementations in Java and C#, with some characteristics of the template system in C++ [4] [8]. In all of these systems, type parameters represent (potentially bounded) unknown runtime types. Differences from the Java and C# systems emerge due to Go's existing type system. Because there is no concept of inheritance in Go, the only way for type parameters to represent multiple potential runtime types is if they are bounded by interface types. Since type parameters bounded by non-interface types would only have a single valid type substitution, they are disallowed in this system. Another difference from the Java and C# systems is that Go's interface system is completely structural rather than nominal, i.e., there is no need to indicate which interfaces a particular type implements. Because of this, the proposed generic system in Go is also structural, and is similar to the concept system used

[1] Slices in Go are dynamic lists of variable size that are backed by arrays

in C++ to describe which types are valid parameters to a class template. The type bound for a type parameter describes which operations must be supported on the runtime type, whereas in C++ this is usually specified by a contract in the documentation or through more complex methods [12]. The C++ compiler essentially performs structural subtype checking when a class template is instantiated with a concrete type, but because the required operations are not explicitly declared in the type system the communication of these requirements to the programmer is less intuitive unless complex concept checking practices are followed. The generic function system described here gains some of the benefits of the structural type bounding, such as the ability to define the required operations of a type without regard to an existing type hierarchy, while also providing the explicit communication of the nominal systems.

5.2 Variance

Go's type system simplifies the implementation of generic functions by removing the need to consider variance in many cases. If a type system contains both parametric polymorphism and subtype polymorphism, additional information must be supplied for each type parameter to indicate how it will interact with the subtype system. Type parameters can take three forms that have different subtype behavior: invariant, covariant, or contravariant. If $<:$ is the subtype relationship, then the following rules apply to the subtype relationship for a type $\texttt{T<S>}$ ($\texttt{T}$ has type parameter $\texttt{S}$):

- If $\texttt{S}$ is invariant, there is no additional subtype relationship:

$$\frac{}{\texttt{T<A>} <: \texttt{T<A>}}$$

- If $\texttt{S}$ is covariant:

$$\frac{\texttt{A} <: \texttt{B}}{\texttt{T<A>} <: \texttt{T<B>}}$$

- If $\texttt{S}$ is contravariant:

$$\frac{\texttt{B} <: \texttt{A}}{\texttt{T<A>} <: \texttt{T<B>}}$$

In Go's type system, the only subtype relationship is the structural subtyping rule for interfaces. Specifically, the interface subtype rule states that

a type `A` implements interface `I` if the methods in `I` are defined on `A`. Since this is the only subtype relationship that exists in Go's type system, any two distinct function types are not subtypes of each other, regardless of the subtype relationships of their arguments or return values.

If the generic system was extended to include generic structs and interfaces, then variance would need to be considered on the type parameters of interfaces. Additional annotations on the type parameter list of generic interfaces would allow the programmer to indicate the variance of each parameter. The type checker would need to ensure that certain rules about the uses of covariant and contravariant type parameters were followed, to ensure that the subtyping rule implied by the variance of the type parameters would result in only type safe operations.

6 Code Generation

The most involved changes to the compiler were required to generate the correct code to implement the new language constructs. Generic functions must be able to accept arguments of types that are different than the types used to type check and generate the body of the function. In the case of arguments that have the direct type of a type parameter, the argument can simply be converted to the interface type of the type bound and used as a regular interface within the body of the function. If a return value from the function uses a type parameter directly, a type assertion can be used after the value is returned to transform it back into the specific type substituted for that type parameter at the call site. Additional steps must be taken for arguments that have more complex types containing type parameters, which will be referred to as complex parameterized types. For these types, such as slices of or pointers to a type parameter, there is no way within the existing type system to translate the memory layout of the actual argument to the expected format. For these values, the code generation system must be extended to provide a method of generating the generic function that is independent of the memory layout of the argument values. There are several possible methodologies for accomplishing this, with trade offs in implementation complexity, runtime speed, and code size.

6.1 AST Transformation

A process of AST transformation is used to translate the input AST into a standard Go AST with no generic functionality. This AST transformation is implemented as a depth-first traversal of the AST, with each node

being transformed by a function into its standard form. In this way, complex expressions are translated by first transforming their parts, and then performing any additional modifications needed to combine the simple expressions into the complex one.

For statements, additional bookkeeping must be done to accommodate transformations that turn one statement into several statements. For instance, this is necessary when dealing with functions with multiple return values. Multiple return values are only accessible through a multi-part assignment statement, so any transformations that need to be made to the individual return values must be made in a separate statement. For example, consider a function `foo` that returns two values, where the AST transformation is represented by a function `f`:

```
a, b := foo()
```

This must be transformed into the two statements:

```
a', b' := foo()
a, b := f(a), f(b)
```

where `a'` and `b'` represent new temporary variables. The AST transformation accommodates this by allowing the transformation function for statements to return a list of result statements, instead of only a single statement. It is left to the surrounding context of the statement to determine the correct way to insert the additional statements. Within blocks, the new statements are simply added to the body of the block, but there are other contexts for statements that require additional thought. There are instances where only a single statement is allowed, such as parts of `if` statements and `for` loops. In these cases, if the statement is transformed into multiple statements, the extra statements must be inserted elsewhere in the AST because only one of the resulting statements can take the place of the original value. For example, consider the case where the previous assignment statement occurred at the beginning of an `if` statement:

```
if a, b := foo(); a != b { ... }
```

This must be transformed as follows:

```
a', b' := foo()
if a, b := f(a), f(b); a != b { ... }
```

If, instead, the assignment statement occurred as the update statement of a `for` loop:

```
for a, b := 0, 0; a == b; a, b = foo() { ... }
```

This would be transformed as:

```
var a', b' T
for a, b := 0, 0; a == b; a, b = f(a'), f(b') {
  ...
  a', b' = foo()
}
```

6.2 Reflection

The code generation method used involves rewriting expressions that include values of complex parameterized types into invocations of the runtime introspection system. The Go standard library provides the `reflect` package, which includes mechanisms for operating on types and values based on their runtime type without knowing the memory layout at compile time. The `reflect` package includes two main types, `reflect.Value` and `reflect.Type`, which provide runtime representations of the value and type sections of interface values. The inputs to the `reflect` package are values of the empty interface type, which all values conform to. Once a value has been transformed into a `reflect.Value`, operations that are normally valid on complex types, such as dereferencing a pointer, indexing a slice, or updating a map, are available as operations on the `reflect.Value`. The results of these reflection expressions can be extracted as an empty interface value, which can optionally be transformed into the concrete expected type using type assertions. In this manner, any value with a complex parameterized type is transformed into an empty interface, and any operations on these values are transformed into reflection operations.

There are many functions in the Go standard library that make use of *de-facto* parametric polymorphism, in that they operate on collection types independent of the element types of the collections. These functions must also be transformed into their reflection equivalents, since all collection values that have type parameters as element types will be represented as empty interface values instead of collection values.

Care must be taken when performing reflection transformations to avoid extracting the value of one reflection operation as an empty interface, only to immediately transform it back into a `reflect.Value`. This is important not only for efficiency, but also in cases where a reflection expression is on the left hand side of an assignment. In this case, the `reflect` package allows certain values to be assignable, but only if they are the result of accessing an addressable part of a value, such as an array element, stuct field, or pointer value. It would not be possible to use the reflection expression generated by transforming the left hand side of such an assignment as an addressable value if it was first extracted as an empty interface and then turned back into a `reflect.Value`. An example of this scenario is the assignment of a

value to a slice index:

```
s[i] = a
```

The naive reflection transformation includes an extra conversion to an empty interface value. When the value is transformed back into a `reflect.Value` it is no longer addressable, and the `Set` operation is invalid on it:

```
temp := reflect.ValueOf(s).Index(i).Interface()
reflect.ValueOf(temp).Set(reflect.ValueOf(value)) //error
```

The correct transformation omits this unnecessary conversion:

```
reflect.ValueOf(s).Index(i).Set(reflect.ValueOf(value))
```

After this transformation is accomplished, the bodies of the generic functions are valid standard Go code, and code generation can continue as normal. Some examples of expressions and their reflection versions follow, to illustrate how the transformation occurs:

- Pointer dereferencing:

  ```
  *p
  ```

  Reflection:

  ```
  reflect.Indirect(reflect.ValueOf(p)).Interface()
  ```

- Slice indexing:

  ```
  s[i]
  ```

  Reflection:

  ```
  reflect.ValueOf(s).Index(i).Interface()
  ```

- Length calculation:

  ```
  len(s)
  ```

  Reflection:

  ```
  reflect.ValueOf(s).Len()
  ```

6.3 Signature Transformation

In addition to transforming the bodies of generic functions, their type signatures and calling expressions must also be changed. Any argument or return types that are complex parameterized types are transformed into empty interface values, so that the correct transformations are made to wrap input values in the empty interface context. Call sites of such functions are updated to perform type assertions on their results, so that the results are returned in the format that the calling expression expects. Additional parameters are also added to the signatures of generic functions. The added parameters are `reflect.Type` values that correspond to specific types substituted for each type parameter at the call site. These values are needed so that allocations involving type parameters can be translated into their reflection versions. For example, a function may include a call to `make` for a slice of a type parameter `A`:

```
func f<A B>() []A {
  return make([]A, 10)
}
```

This function will be transformed into:

```
func f<A B>(t$A reflect.Type) interface{} {
  return reflect.MakeSlice(t$A, 10)
}
```

If the reflection version of `make` was not used in this case, the returned slice would always be a slice of the type bound of `A`, not a slice of the concrete type supplied at the call site. A type assertion at the call sites of `f` transforms the resulting empty interface value into the slice of the concrete type expected.

6.4 Alternatives for Code Generations

The reflection method of generating generic code is only one implementation possibility. The reflection system makes use of the structure of interface values internally. Each interface value consists of two pointers, one to the data contained within the value, and another to the type information for the runtime concrete type of the value. The reflection system operates on these type and value pointers, performing the necessary transformations to each to produce the next set of values. Using the reflection system directly involves many function calls, as well as runtime checks to ensure that the operations performed on the reflection values are valid given the actual runtime type of the input. In the case of generic code generation, these calls

could be avoided by performing the operations on the type and value pointers directly without invoking a library. Many of the runtime checks could potentially be avoided as well, since the type system should ensure that the input values have the correct form before the reflection operations take place. This inline introspection could provide performance improvements over the AST transformations used here. They would require significant changes to the lower level parts of the code generation system, at both the SSA and LLVM IR levels. The current system has the advantage that since it does not change the low level code generation system, it cannot introduce bugs in these complex systems. For a proof of concept, the advantages in ease of development and debugging were chosen over the potential performance gains.

Another code generation possibility would avoid use of introspection completely. The need for introspection arises because the memory layout of values of different instantiations of a generic function are different, and the body of a generic function must be able to handle values with many different memory layouts. One way to avoid this complication is to create distinct versions of each function body for all of the different concrete instantiations of the function that are used in the program. This is more similar to the template system in C++, and involves duplicating generic code to produce non-generic versions that operate normally. This technique has the performance advantage of allowing static dispatch of methods called on values that would otherwise require dynamic dispatch using the interface system. Statically dispatched methods can more easily be optimized or inlined at the call site. In addition to static dispatch, specialized versions of generic functions can allocate more values on the stack or in registers because the memory layout is fixed and known at compile time, which can expose additional optimizations.

The drawbacks of specialization are in code size and compilation time. A copy of each generic function must be generated and compiled for every combination of type variables that is used in the program. The existence of function pointers compounds this drawback. When pointers to generic functions are used, it is no longer possible in general to determine the minimal set types that are used as arguments to a function. Pointer analysis can reduce the number of functions that a function pointer can alias, but in the worst case it is possible to have function pointers that can alias any function that matches their signature. This means that any calls to that function pointer produce copies of every function with a matching signature, which may then cause additional functions to be generated if the bodies of those functions themselves call generic functions. Function pointers also require

runtime changes to code generation, as any function pointer with a generic signature must now refer to the table of all instantiations of the function that the pointer references. This allows the eventual call site to select the correct instantiation based on the type arguments.

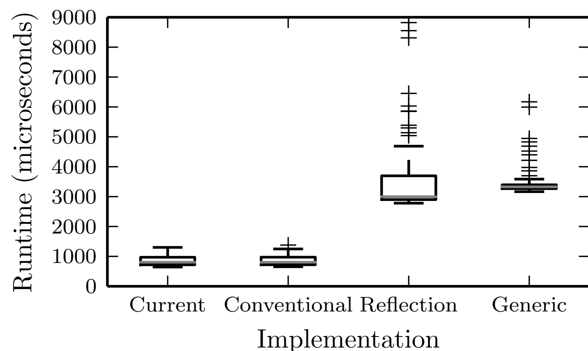
Since specialization and introspection both have drawbacks, in code size and performance respectively, a hybrid approach may be the most effective. Heuristics or runtime information could be used to determine which functions are called most often so that they could be partially specialized for the most common type arguments. Compiler hints could also be used by the programmer to indicate which functions would likely benefit from specialization. The hybrid approach has the benefit of optimizing the hot paths without introducing significant code bloat. This method could gain the benefits of both code generation methodologies, but it would require all of the implementation work for building both methods and the heuristics needed to choose the generation strategy.

7 Performance

The use of reflection to implement polymorphism does incur a significant runtime cost. Two benchmarks are presented below, with four implementations compared. The first and second implementations are identical, but the first is compiled using the existing LLGO compiler, while the second is compiled with the modified version. Both versions are included to demonstrate that non-generic code is unaffected by the modifications to the language. The first and second implementations are conventional functions that can only operate on a single type and have no polymorphism. The third implementation uses hand-written reflection operations to provide support for polymorphism, but it does not use the generic system. The final implementation uses the generic system. The two benchmarks represent functions that generate different proportions of their code as reflection operations.

The first benchmark, in Figure 1, is an implementation of a `map` function, which applies a function to each element in a slice in order to produce a new slice. The `map` function is a staple of functional programming techniques, and it benefits greatly from the type safety introduced by parametric polymorphism. For the benchmark, the function that is used in the `map` operation is the factorial function, which is used as a placeholder for any function that performs significant computation on the values within the slice. The `map` function performs relatively little manipulation of complex parameterized types, as each element is retrieved once from one slice and set once in an-

Figure 1: Map Runtime



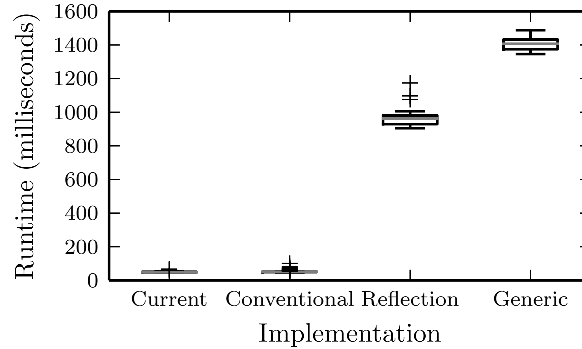
other. The `map` function was run on lists of 10,000 elements ranging from 0 to 20. The hand-written reflection implementation was 3.72 times slower on average than the conventional version. The generic version was 4.13 times slower on average than the conventional version, but only 11% slower than the hand-written reflection implementation.

The second benchmark, in Figure 2, is a naive `quicksort` function, which sorts an input slice. The `quicksort` function accepts a predicate as an argument that compares two values, so that collections of arbitrary types may be sorted. The `quicksort` algorithm is close to a worst case scenario for this generic implementation, because in contrast to `map`, most of the computational work in the `quicksort` implementation is directly working with complex parametric types. Each comparison and swap requires index operations to access and set the values within the slice, and each recursive call to `quicksort` involves creating a new slice using an interval of the input slice. The `quicksort` function was called with a list of 10,000 random integers, and a list of 10,000 random floating point values. The reflection implementation was 18.88 times slower than the conventional version. The generic implementation was 27.59 times slower than the conventional version, and 1.46 times slower than the hand-written reflection implementation.

Both reflection implementations have additional overhead compared to the conventional version due to the allocations that must be performed to create the intermediate reflection values and operate on them. The hand-written reflection implementation can reduce duplication of reflection operations by reusing values in multiple expressions. Profiling data for the hand-written reflection implementation[2] in Table 1 indicates that most of

[2] Profiling is not currently available for generic functions

Figure 2: Quicksort Runtime



the additional execution time is spent within the allocation code in the `reflect` package, particularly when extracting values out of the reflection context after an operation completes. The conventional implementation also has the advantage of allowing many operations to be statically allocated, or to use registers. More optimization opportunities are exposed to the compiler in the conventional versions for this reason.

These results indicate that the runtime cost associated with generic functions is significant in this implementation. Performing the introspection directly instead of dispatching to the existing reflection system could produce performance gains by avoiding the allocation of intermediate objects. These results also suggest that generating specialized versions of generic function implementations could provide significant performance improvements, as the specialized functions generated would be closer to the hand-written conventional functions that served as the baseline for these benchmarks. Since the performance of non-generic code is unaffected by the generic system, the programmer can determine if the abstraction provided by generic programming is worth the trade off in speed.

8 Conclusion and Future Work

The generic function implementation described demonstrates that parametric polymorphism is a concept that is compatible with Go. Two of Go's core philosophies are simplicity and ease of use, and the extensions to the language required to add generics do not violate these principles. The syntax extensions are not substantially harder to understand or use than existing language constructs, and the type system is still less complex than com-

Time	Time (%)	Function
8.83s	76.92%	main.quicksortReflect
5.09s	44.34%	reflect.Value.Interface
4.80s	41.81%	reflect.valueInterface
4.38s	38.15%	reflect.packEface
3.07s	26.74%	runtime.newobject
2.91s	25.35%	reflect.unsafe_New
2.73s	23.78%	runtime.mallocgc
1.18s	10.28%	runtime.typedmemmove
1.03s	8.97%	reflect.typedmemmove
0.85s	7.40%	reflect.Value.Set
0.80s	6.97%	runtime.memmove
0.62s	5.40%	runtime.assertE2T
0.59s	5.14%	reflect.Value.Index

Table 1: Cumulative runtime of functions in the hand-written implementation of `quicksort`. Only functions called from within the sorting code and that account for at least 5% of the total runtime are included. The `runtime/pprof` package of the Go standard library was used to collect performance data.

parable languages like C++ or Java due to the lack of inheritance. The additional complexity introduced in the type system is balanced by a reduction in complexity in user code and increased expressiveness for writing reusable programs. This implementation provides a proof of concept for how a generic type system might interface with existing Go language constructs, and demonstrates that the language can be extended in a backwards compatible way.

An extension to the generic type system would be to allow for multiple type bounds for each type parameter. This can be emulated for non-overlapping interface type bounds in the current implementation by constructing a new interface type that has all of the desired type bounds as embedded interfaces. If the type bounds have overlapping method sets, a new interface must be constructed that contains the union of all of the method sets of the type bounds. In the future, this step could be performed automatically for all supplied type bounds.

The runtime costs incurred by this implementation of generic functions are significant, and do challenge Go's focus on performance. Several techniques for improving the performance of generic code have been discussed, and future work can evaluate if one of these proposals provides satisfactory

performance trade offs. The changes needed to evaluate the performance of function specialization or inline introspection are limited to the code generation section of the compiler, and the existing infrastructure for parsing and type checking the generic language constructs can be reused. The performance penalties of this particular implementation of the generic function system do not necessarily indicate that the language extension is incompatible with a performance oriented language, only that additional work is needed to satisfy this design goal.

A complete implementation of parametric polymorphism will need to contain mechanisms for generic interfaces and structs. Generic functions are a necessary first step to implementing these other generic constructs, as both interfaces and structs are largely defined by their member functions. The principles used in parsing and type checking generic functions are extendable to the other language constructs, and some parts of the implementation may be reused.

Finally, if a complete, performant generic type implementation is developed it will need to be implemented in the major Go compilers, GC and GCCGo. The LLGO compiler is useful in developing the prototype system and for more rapid development of alternative implementations for performance comparisons, but the main Go compilers are what define the language in practice.

The source code for the modified version of the LLGO compiler can be found at `https://github.com/Matt343/llgo`.

References

[1] Cardelli, L. Basic polymorphic typechecking. *Science of computer programming 8*, 2 (1987), 147–172.

[2] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 4 (1991), 451–490.

[3] Free Software Foundation, I. Gcc, the gnu compiler collection. `https://gcc.gnu.org/`. Accessed: 2016-04-19.

[4] Ghosh, D. Generics in java and c++: a comparative model. *ACM SIGPLAN Notices 39*, 5 (2004), 40–47.

[5] Google. Frequently asked questions (faq). `https://golang.org/doc/faq`. Accessed: 2016-04-19.

[6] Google. Go 1.5 release notes. `http://tip.golang.org/doc/go1.5`. Accessed: 2016-04-19.

[7] Google. The go programming language specification. `https://golang.org/ref/spec`. Accessed: 2016-04-19.

[8] Kennedy, A., and Syme, D. Design and implementation of generics for the. net common language runtime. In *ACM SigPlan Notices* (2001), vol. 36, ACM, pp. 1–12.

[9] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (2004), IEEE, pp. 75–86.

[10] Milner, R. A theory of type polymorphism in programming. *Journal of computer and system sciences 17*, 3 (1978), 348–375.

[11] Pike, R. Go at google. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity* (2012), ACM, pp. 5–6.

[12] Siek, J., and Lumsdaine, A. Concept checking: Binding parametric polymorphism in c++. In *First Workshop on C++ Template Programming* (2000), Germany.

Appendices

A Modified Grammar

The complete Go grammar is included below. Non-terminals are enclosed in brackets: ⟨*NonTerminal*⟩. Terminal strings are enclosed in single quotes: ‘`terminal`’

A.1 New or Modified Productions

⟨*Signature*⟩ ::= [⟨*TypeParams*⟩] ⟨*Parameters*⟩ [⟨*Result*⟩]

⟨*TypeParams*⟩ ::= ‘<’ ⟨*TypeParamList*⟩ [‘,’] ‘>’

⟨*TypeParamList*⟩ ::= ⟨*TypeParamDecl*⟩ { ‘,’ ⟨*TypeParamDecl*⟩ }

⟨*TypeParamDecl*⟩ ::= ⟨*IdentifierList*⟩ ⟨*Type*⟩

⟨*Arguments*⟩ ::= ‘(’ [⟨*TypeArgumens*⟩ ‘,’] [⟨*ValueArguments*⟩] ‘)’

⟨*TypeArguments*⟩ ::= ‘<’ ⟨*TypeList*⟩ [‘,’] ‘>’

⟨*TypeList*⟩ ::= ⟨*Type*⟩ { ‘,’ ⟨*Type*⟩ }

⟨*ValueArguments*⟩ ::= ⟨*AgumentList*⟩ [‘...’] [‘,’]

⟨*ArgumentList*⟩ ::= ⟨*ExpressionList*⟩
 | ⟨*Type*⟩ [‘,’ ⟨*ExpressionList*⟩]

A.2 Unchanged Productions

⟨*Type*⟩ ::= ⟨*TypeName*⟩
 | ⟨*TypeLit*⟩
 | ‘(’ ⟨*Type*⟩ ‘)’

⟨*TypeName*⟩ ::= ⟨*identifier*⟩ | ⟨*QualifiedIdent*⟩

⟨*TypeLit*⟩ ::= ⟨*ArrayType*⟩ | ⟨*StructType*⟩ | ⟨*PointerType*⟩ | ⟨*FunctionType*⟩
 | ⟨*InterfaceType*⟩ | ⟨*SliceType*⟩ | ⟨*MapType*⟩ | ⟨*ChannelType*⟩

⟨*ArrayType*⟩ ::= ‘[’ ⟨*ArrayLength*⟩ ‘]’ ⟨*ElementType*⟩

⟨*ArrayLength*⟩ ::= ⟨*Expression*⟩

⟨*ElementType*⟩ ::= ⟨*Type*⟩

⟨*SliceType*⟩ ::= '`[`' '`]`' ⟨*ElementType*⟩

⟨*StructType*⟩ ::= '`struct`' '`{`' { ⟨*FieldDecl*⟩ '`;`' } '`}`'

⟨*FieldDecl*⟩ ::= (⟨*IdentifierList*⟩ ⟨*Type*⟩ | ⟨*AnonymousField*⟩) [⟨*Tag*⟩]

⟨*AnonymousField*⟩ ::= ['`*`'] ⟨*TypeName*⟩

⟨*Tag*⟩ ::= ⟨*string_lit*⟩

⟨*PointerType*⟩ ::= '`*`' ⟨*BaseType*⟩

⟨*BaseType*⟩ ::= ⟨*Type*⟩

⟨*FunctionType*⟩ ::= '`func`' ⟨*Signature*⟩

⟨*Result*⟩ ::= ⟨*Parameters*⟩ | ⟨*Type*⟩

⟨*Parameters*⟩ ::= '`(`' [⟨*ParameterList*⟩ ['`,`']] '`)`'

⟨*ParameterList*⟩ ::= ⟨*ParameterDecl*⟩ { '`,`' ⟨*ParameterDecl*⟩ }

⟨*ParameterDecl*⟩ ::= [⟨*IdentifierList*⟩] ['`...`'] ⟨*Type*⟩

⟨*InterfaceType*⟩ ::= '`interface`' '`{`' { ⟨*MethodSpec*⟩ '`;`' } '`}`'

⟨*MethodSpec*⟩ ::= ⟨*MethodName*⟩ ⟨*Signature*⟩
| ⟨*InterfaceTypeName*⟩

⟨*MethodName*⟩ ::= ⟨*identifier*⟩

⟨*InterfaceTypeName*⟩ ::= ⟨*TypeName*⟩

⟨*MapType*⟩ ::= '`map`' '`[`' ⟨*KeyType*⟩ '`]`' ⟨*ElementType*⟩

⟨*KeyType*⟩ ::= ⟨*Type*⟩

⟨*ChannelType*⟩ ::= ('`chan`' | '`chan`' '`<-`' | '`<-`' '`chan`') ⟨*ElementType*⟩

⟨*Block*⟩ ::= '`{`' ⟨*StatementList*⟩ '`}`'

⟨*StatementList*⟩ ::= { ⟨*Statement*⟩ ‘`;`’ }

⟨*Declaration*⟩ ::= ⟨*ConstDecl*⟩ | ⟨*TypeDecl*⟩ | ⟨*VarDecl*⟩

⟨*TopLevelDecl*⟩ ::= ⟨*Declaration*⟩ | ⟨*FunctionDecl*⟩ | ⟨*MethodDecl*⟩

⟨*ConstDecl*⟩ ::= ‘`const`’ (⟨*ConstSpec*⟩ | ‘`(`’ { ⟨*ConstSpec*⟩ ‘`;`’ } ‘`)`’)

⟨*ConstSpec*⟩ ::= ⟨*IdentifierList*⟩ [[⟨*Type*⟩] ‘`=`’ ⟨*ExpressionList*⟩]

⟨*IdentifierList*⟩ ::= ⟨*identifier*⟩ { ‘`,`’ ⟨*identifier*⟩ }

⟨*ExpressionList*⟩ ::= ⟨*Expression*⟩ { ‘`,`’ ⟨*Expression*⟩ }

⟨*TypeDecl*⟩ ::= ‘`type`’ (⟨*TypeSpec*⟩ | ‘`(`’ { ⟨*TypeSpec*⟩ ‘`;`’ } ‘`)`’)

⟨*TypeSpec*⟩ ::= ⟨*identifier*⟩ ⟨*Type*⟩

⟨*VarDecl*⟩ ::= ‘`var`’ (⟨*VarSpec*⟩ | ‘`(`’ { ⟨*VarSpec*⟩ ‘`;`’ } ‘`)`’)

⟨*VarSpec*⟩ ::= ⟨*IdentifierList*⟩
    (⟨*Type*⟩ [‘`=`’ ⟨*ExpressionList*⟩] | ‘`=`’ ⟨*ExpressionList*⟩)

⟨*ShortVarDecl*⟩ ::= ⟨*IdentifierList*⟩ ‘`:=`’ ⟨*ExpressionList*⟩

⟨*FunctionDecl*⟩ ::= ‘`func`’ ⟨*FunctionName*⟩ (⟨*Function*⟩ | ⟨*Signature*⟩)

⟨*FunctionName*⟩ ::= ⟨*identifier*⟩

⟨*Function*⟩ ::= ⟨*Signature*⟩ ⟨*FunctionBody*⟩

⟨*FunctionBody*⟩ ::= ⟨*Block*⟩

⟨*MethodDecl*⟩ ::= ‘`func`’ ⟨*Receiver*⟩ ⟨*MethodName*⟩
    (⟨*Function*⟩ | ⟨*Signature*⟩)

⟨*Receiver*⟩ ::= ⟨*Parameters*⟩

⟨*Operand*⟩ ::= ⟨*Literal*⟩
  | ⟨*OperandName*⟩
  | ⟨*MethodExpr*⟩
  | ‘`(`’ ⟨*Expression*⟩ ‘`)`’

⟨Literal⟩ ::= ⟨BasicLit⟩ | ⟨CompositeLit⟩ | ⟨FunctionLit⟩

⟨BasicLit⟩ ::= ⟨int_lit⟩ | ⟨float_lit⟩ | ⟨imaginary_lit⟩ | ⟨rune_lit⟩ | ⟨string_lit⟩

⟨OperandName⟩ ::= ⟨identifier⟩ | ⟨QualifiedIdent⟩

⟨QualifiedIdent⟩ ::= ⟨PackageName⟩ '.' ⟨identifier⟩

⟨CompositeLit⟩ ::= ⟨LiteralType⟩ ⟨LiteralValue⟩

⟨LiteralType⟩ ::= ⟨StructType⟩
  | ⟨ArrayType⟩
  | '[' '...' ']' ⟨ElementType⟩
  | ⟨SliceType⟩
  | ⟨MapType⟩
  | ⟨TypeName⟩

⟨LiteralValue⟩ ::= '{' [⟨ElementList⟩ [',']] '}'

⟨ElementList⟩ ::= ⟨KeyedElement⟩ { ',' ⟨KeyedElement⟩ }

⟨KeyedElement⟩ ::= [⟨Key⟩ ':'] ⟨Element⟩

⟨Key⟩ ::= ⟨FieldName⟩ | ⟨Expression⟩ | ⟨LiteralValue⟩

⟨FieldName⟩ ::= ⟨identifier⟩

⟨Element⟩ ::= ⟨Expression⟩ | ⟨LiteralValue⟩

⟨FunctionLit⟩ ::= 'func' ⟨Function⟩

⟨PrimaryExpr⟩ ::= ⟨Operand⟩
  | ⟨Conversion⟩
  | ⟨PrimaryExpr⟩ ⟨Selector⟩
  | ⟨PrimaryExpr⟩ ⟨Index⟩
  | ⟨PrimaryExpr⟩ ⟨Slice⟩
  | ⟨PrimaryExpr⟩ ⟨TypeAssertion⟩
  | ⟨PrimaryExpr⟩ ⟨Arguments⟩

⟨Selector⟩ ::= '.' ⟨identifier⟩

⟨Index⟩ ::= '[' ⟨Expression⟩ ']'

⟨*Slice*⟩ ::= ‘[’ ⟨*SliceIndices*⟩ ‘]’

⟨*SliceIndices*⟩ ::= ([⟨*Expression*⟩] ‘:’ [⟨*Expression*⟩])
  | ([⟨*Expression*⟩] ‘:’ ⟨*Expression*⟩ ‘:’ ⟨*Expression*⟩)

⟨*TypeAssertion*⟩ ::= ‘.’ ‘(’ ⟨*Type*⟩ ‘)’

⟨*MethodExpr*⟩ ::= ⟨*ReceiverType*⟩ ‘.’ ⟨*MethodName*⟩

⟨*ReceiverType*⟩ ::= ⟨*TypeName*⟩
  | ‘(’ ‘*’ ⟨*TypeName*⟩ ‘)’
  | ‘(’ ⟨*ReceiverType*⟩ ‘)’

⟨*Expression*⟩ ::= ⟨*UnaryExpr*⟩
  | ⟨*Expression*⟩ ⟨*binary_op*⟩ ⟨*Expression*⟩

⟨*UnaryExpr*⟩ ::= ⟨*PrimaryExpr*⟩
  | ⟨*unary_op*⟩ ⟨*UnaryExpr*⟩

⟨*binary_op*⟩ ::= ‘||’ | ‘&&’ | ⟨*rel_op*⟩ | ⟨*add_op*⟩ | ⟨*mul_op*⟩

⟨*rel_op*⟩ ::= ‘==’ | ‘!=’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’

⟨*add_op*⟩ ::= ‘+’ | ‘-’ | ‘|’ | ‘^’

⟨*mul_op*⟩ ::= ‘*’ | ‘/’ | ‘%’ | ‘<<’ | ‘>>’ | ‘&’ | ‘&^’

⟨*unary_op*⟩ ::= ‘+’ | ‘-’ | ‘!’ | ‘^’ | ‘*’ | ‘&’ | ‘<-’

⟨*Conversion*⟩ ::= ⟨*Type*⟩ ‘(’ ⟨*Expression*⟩ [‘,’] ‘)’

⟨*Statement*⟩ ::= ⟨*Declaration*⟩ | ⟨*LabeledStmt*⟩ | ⟨*SimpleStmt*⟩ | ⟨*GoStmt*⟩ |
  ⟨*ReturnStmt*⟩ | ⟨*BreakStmt*⟩ | ⟨*ContinueStmt*⟩ | ⟨*GotoStmt*⟩ | ⟨*FallthroughStmt*⟩
  | ⟨*Block*⟩ | ⟨*IfStmt*⟩ | ⟨*SwitchStmt*⟩ | ⟨*SelectStmt*⟩ | ⟨*ForStmt*⟩ | ⟨*DeferStmt*⟩

⟨*SimpleStmt*⟩ ::= ⟨*ExpressionStmt*⟩ | ⟨*SendStmt*⟩ | ⟨*IncDecStmt*⟩ | ⟨*Assignment*⟩
  | ⟨*ShortVarDecl*⟩

⟨*LabeledStmt*⟩ ::= ⟨*Label*⟩ ‘:’ ⟨*Statement*⟩

⟨*Label*⟩ ::= ⟨*identifier*⟩

⟨*ExpressionStmt*⟩ ::= ⟨*Expression*⟩

⟨*SendStmt*⟩ ::= ⟨*Channel*⟩ '<-' ⟨*Expression*⟩

⟨*Channel*⟩ ::= ⟨*Expression*⟩

⟨*IncDecStmt*⟩ ::= ⟨*Expression*⟩ ('++' | '--')

⟨*Assignment*⟩ ::= ⟨*ExpressionList*⟩ ⟨*assign_op*⟩ ⟨*ExpressionList*⟩

⟨*assign_op*⟩ ::= [⟨*add_op*⟩ | ⟨*mul_op*⟩] '='

⟨*IfStmt*⟩ ::= 'if' [⟨*SimpleStmt*⟩ ';'] ⟨*Expression*⟩ ⟨*Block*⟩ [⟨*ElseStmt*⟩]

⟨*ElseStmt*⟩ ::= 'else' (⟨*IfStmt*⟩ | ⟨*Block*⟩)

⟨*SwitchStmt*⟩ ::= ⟨*ExprSwitchStmt*⟩ | ⟨*TypeSwitchStmt*⟩

⟨*ExprSwitchStmt*⟩ ::= 'switch' [⟨*SimpleStmt*⟩ ';'] [⟨*Expression*⟩]
'{' { ⟨*ExprCaseClause*⟩ } '}'

⟨*ExprCaseClause*⟩ ::= ⟨*ExprSwitchCase*⟩ ':' ⟨*StatementList*⟩

⟨*ExprSwitchCase*⟩ ::= 'case' ⟨*ExpressionList*⟩
| 'default'

⟨*TypeSwitchStmt*⟩ ::= 'switch' [⟨*SimpleStmt*⟩ ';'] ⟨*TypeSwitchGuard*⟩
'{' { ⟨*TypeCaseClause*⟩ } '}'

⟨*TypeSwitchGuard*⟩ ::= [⟨*identifier*⟩ ':='] ⟨*PrimaryExpr*⟩ '.' '(' 'type' ')'

⟨*TypeCaseClause*⟩ ::= ⟨*TypeSwitchCase*⟩ ':' ⟨*StatementList*⟩

⟨*TypeSwitchCase*⟩ ::= 'case' ⟨*TypeList*⟩
| 'default'

⟨*ForStmt*⟩ ::= 'for' [⟨*Condition*⟩ | ⟨*ForClause*⟩ | ⟨*RangeClause*⟩] ⟨*Block*⟩

⟨*Condition*⟩ ::= ⟨*Expression*⟩

⟨*ForClause*⟩ ::= [⟨*InitStmt*⟩] ';' [⟨*Condition*⟩] ';' [⟨*PostStmt*⟩]

⟨*InitStmt*⟩ ::= ⟨*SimpleStmt*⟩

⟨*PostStmt*⟩ ::= ⟨*SimpleStmt*⟩

⟨*RangeClause*⟩ ::= [⟨*ExpressionList*⟩ ‘`=`’ | ⟨*IdentifierList*⟩ ‘`:=`’] ‘`range`’ ⟨*Expression*⟩

⟨*GoStmt*⟩ ::= ‘`go`’ ⟨*Expression*⟩

⟨*SelectStmt*⟩ ::= ‘`select`’ ‘`{`’ { ⟨*CommClause*⟩ } ‘`}`’

⟨*CommClause*⟩ ::= ⟨*CommCase*⟩ ‘`:`’ ⟨*StatementList*⟩

⟨*CommCase*⟩ ::= ‘`case`’ (⟨*SendStmt*⟩ | ⟨*RecvStmt*⟩) | ‘`default`’

⟨*RecvStmt*⟩ ::= [⟨*ExpressionList*⟩ ‘`=`’ | ⟨*IdentifierList*⟩ ‘`:=`’] ⟨*RecvExpr*⟩

⟨*RecvExpr*⟩ ::= ⟨*Expression*⟩

⟨*ReturnStmt*⟩ ::= ‘`return`’ [⟨*ExpressionList*⟩]

⟨*BreakStmt*⟩ ::= ‘`break`’ [⟨*Label*⟩]

⟨*ContinueStmt*⟩ ::= ‘`continue`’ [⟨*Label*⟩]

⟨*GotoStmt*⟩ ::= ‘`goto`’ ⟨*Label*⟩

⟨*FallthroughStmt*⟩ ::= ‘`fallthrough`’

⟨*DeferStmt*⟩ ::= ‘`defer`’ ⟨*Expression*⟩

⟨*SourceFile*⟩ ::= ⟨*PackageClause*⟩ ‘`;`’ { ⟨*ImportDecl*⟩ ‘`;`’ }
    { ⟨*TopLevelDecl*⟩ ‘`;`’ }

⟨*PackageClause*⟩ ::= ‘`package`’ ⟨*PackageName*⟩

⟨*PackageName*⟩ ::= ⟨*identifier*⟩

⟨*ImportDecl*⟩ ::= ‘`import`’ (⟨*ImportSpec*⟩ | ‘`(`’ { ⟨*ImportSpec*⟩ ‘`;`’ } ‘`)`’)

⟨*ImportSpec*⟩ ::= [‘`.`’ | ⟨*PackageName*⟩] ⟨*ImportPath*⟩

⟨*ImportPath*⟩ ::= ⟨*string_lit*⟩