Milestone 2

Guillaume Labranche William Bain Si Mei Zhang

March 6, 2015

Contents

1 Design decisions

1.1 Typechecker

In accordance with the instructions our type checking is executed in a single pass without forward declaration.[1] We are working in Java and we implemented the typechecker as a subclass of the `DepthFirstAdapter` class provided by SableCC. This made it easy to traverse the AST without extra boilerplate. Since we did not have a reliable way of editing the extending the AST classes generated by SableCC, we stored type information for the AST in a hash table mapping from nodes to Go types.[2]

In most cases it was sufficient to apply typechecks after the the typechecker had recursed over the child nodes and typechecked them. There were two cases where more fine-grained control was necessary:

- In short assignment statements, the list of variables is implemented as a list of expressions for reasons having to do with the parser implementation. It was therefore necessary to stop the typechecker from typechecking the variables before they had been added to the symbol table.
- To typecheck function declarations and if statements, it was necessary to open a scope after having typechecked some but not all of the child nodes.

In each of these special cases we overrode the `case` method which controls the recursion over the child nodes.

A further implementation detail that is worth noting is our implementation of struct type declarations. To prevent code duplication, we treat struct fields as if they were variable declarations: we open a scope in the symbol table upon entering the struct and enter each field as if it were a variable. Then we pop the scope and use it to build the struct class.

1.2 Symbol table

Initially the symbol table was simply a scope with a reference to the parent scope. Traversing up and down the scopes meant overwriting the variable used to reference the scope, so we decided to make `SymbolTable` be what its name suggests, and take care of scoping up and down. It holds a double-ended queue (Java's `ArrayDeque`) to store the scopes. We have methods for different use cases: searching for an identifier through all scope levels, and searching only in the current scope.

We have also added loggers to enable the `-dumpsymtab` and `-dumpsymtaball` CLI options.

1.3 Type and symbol class hierarchy

We put considerable effort into developing and revising the hierarchy representing GoLite types and symbols (figure 1). We adopted the following goals, listed here roughly in descending order of precedence:

1. Type safety: as much as possible, it should not be possible to use an object in a place where it is not allowed. This should be detected at compile time.
2. Simplicity: there should not be more classes or objects than are necessary. For instance, we did not want to use a class `AliasTypeSymbol(Symbol s)` to wrap every type alias entered in the symbol table.
3. DRY: it should not be necessary to implement functionality in multiple places.

These considerations led us to a number of particular decisions. First, we place functions outside the type hierarchy. While in Go functions are first class citizens, with their type given by the function signature, in GoLite they are not. Therefore, instead of implementing functions as instances of `Variable`, which would potentially allow them to be used where another objects of another type are required, we implemented

[1] See https://mycourses2.mcgill.ca/d2l/le/161312/discussions/threads/258597/View.

[2] By extending I mean adding methods and properties to a class used by the parser (here probably `PExp`, the abstract expression production class), not just creating a subclass.

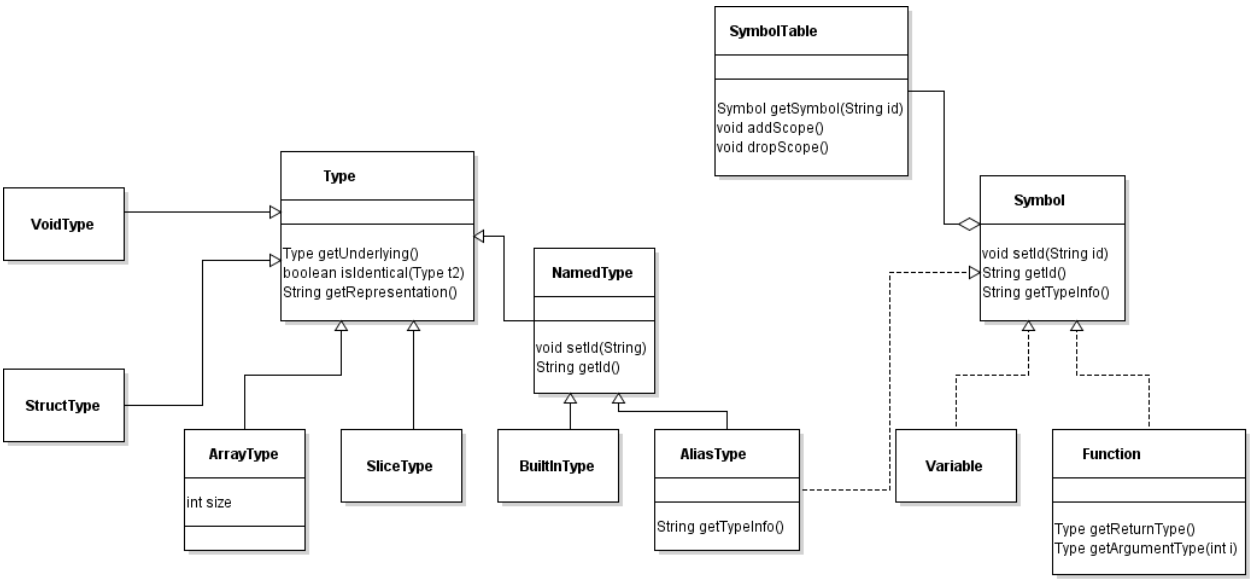

Figure 1: Class hierarchy for GoLite types and symbols (property and method listings are non-exhaustive)

a `Function` class which represents a particular function and stores both the function's identifier and its associated type information directly.

Second, we made `Symbol` an interface which is implemented by the `Variable`, `Function`, and `AliasType` classes. The difficulty here was that since `AliasType` is a subclass of `Type`, it could not also inherit from an abstract `Symbol` class. By making `Symbol` an interface, we allow `AliasType` instances into the symbol table directly without thereby making it possible to enter other types—for instance the `BuiltInType` which we use for `int`, `bool`, `alias`, and `rune`—illicitly. The downside of this design is that the `getId` and `setId` methods required by `Symbol` had to be implemented at least twice. However, this was sufficiently trivial that it did not change our decision. In fact, the duplication is so trivial that we did not even create a shared superclass for `Function` and `Variable` to reduce the duplication there.[3]

A final decision worth noting is our use of the `VoidType` class. Since unfortunately Java does not make it possible to statically forbid null pointers, it is easy to introduce them accidentally. To help catch such cases, we use `VoidType` instead of `null` to represent the return type of a function that does not return a value.

1.4 Typed pretty printer

We extended the `PrettyPrinter` class we implemented in Milestone 1 to have it print extra type information. By overriding the `defaultOut` method, it was simple to have it look up and output the type of any node which is an instance of the `PExp` (expression production) class as an inline comment.

2 Scoping rules

The scoping rules are in accordance with the GoLite specifications:

- In a given scope, it is legal to refer to an identifier that was previously declared in the scope or one of the parent scopes.
- It is illegal to declare a new type, variable or function using an identifier previously defined in the current scope only.

[3] We *did*, however, create a `NamedType` class which is extended by both `BuiltInType` and `AliasType`. This class also implements the `isIdentical` and `getRepresentation` methods.

- The following language structures create new scopes for themselves: `if`, `else`, `for`, `func`, `struct`, standalone block (`{...}`). The if and else branches of the if statement also create their own scopes.

3 Type check operations

TODO description and concordance

In some cases there are multiple code paths which lead to the same error. For instance, the helper method `ensureUndeclared(TId id)` is called in a number of places to ensure that an identifier has not previously been declared in the current scope. While we have done fuzz testing, we have not created an exhaustive suite of failing programs to cover each possible path to the error in such cases.

3.1 General

Use of undefined variable: `identifier_undefined_in_symtab.go`, `append_id_undefined.go`

Redeclaration of variable in same scope: `redeclaration_of_var_in_same_scope.go`

Mismatching types in binary operation: Generic check that operands share the same type. `binary_op_mismatch.go`, `assign_op.go`

Mismatching types in op-assign statement: Check that operands are of a type the operation supports. `field_selection_id_mismatched.go`

3.2 Toplevel declarations

3.3 Statements

3.3.1 Assignment

Assignment to a variable with different underlying basic type: `use_different_type_in_assignment.go`

3.3.2 Op assignment

***

3.3.3 Increment/decrement statement

***

3.3.4 Short variable declaration

***

3.3.5 Variable declaration

***

3.3.6 Type declaration

***

3.3.7 Print statement

***

3.3.8 Return statement

***

3.3.9 If statement

***

3.3.10 Switch statement

***

3.3.11 For statement

***

3.4 Expressions

3.4.1 Variable reference

***

3.4.2 Array access

***

3.4.3 Field access

Field not in struct type: `field_missing.go`

Expression is not a struct: `field_from_non_struct.go`

3.4.4 Function call

***

3.4.5 Type cast

***

3.4.6 Append

Append to a slice of the wrong type: `append_expr_bad_type.go`

Append to a non-slice variable: `append_id_not_slice.go`

3.4.7 Operator expressions

Operands should be of a type supported by the operator :
`binary_op_unsupported_type.go`

Operands should have the same type: Covering a number of operators and types...
`field_selection_id_mismatched_type.go`
`binary_expr_mismatched_types1.go`
`binary_expr_mismatched_types2.go`
`binary_expr_mismatched_types3.go`
`binary_expr_mismatched_types4.go`

```
binary_expr_mismatched_types5.go
binary_expr_mismatched_types6.go
binary_expr_mismatched_types7.go
```

3.4.8 Unary expression

***

4 Breakdown of major contributions

Guillaume Labranche Type checker architecture, symbol table implementation, type checker implementation (especially toplevel declarations and types)

William Bain Type checker architecture, command-line interface, type checker implementation (especially expressions and types)

Si Mei Zhang Type checker implementation (especially statements), test programs

5 Known issues

We have not yet fully implemented blank identifiers in the parser, with the result that it is possible for illegal occurences inside expressions to make it through. However, a program which uses them legally should typecheck correctly.