Milestone2

Yiyang Liu, Yue Lyu, Chunyao Zhao

March 2020

1 Introduction

In this milestone, we constructed a symbol table and a type checker, which type checked the program and associated each identifier to its declaration in one pass of AST. When traversing the AST, we first ensured each declaration type checked and then added the corresponding symbol into our symbol table. The type checker was responsible for typechecking the expressions, and it would infer the type for each expression based on the symbol table. The statements in GoLite programs were typechecked by virtue of our implementation of type checker of expressions.

Since each variable must be first declared before being used, our implementation traversed the AST only once. It dynamically checked and inferred the type for each expression, and added the symbol for each declaration into the scoped symbol table. This design was highly runtime-efficient and would reduce the execution latency of our compiler .

We also reconstructed our tree structure. Our tree now has 5 node kinds, program, declaration, type, statement, and expression. We used the node kind type extensively in our type checker to store and compare the types of the elements in the program.

2 Major Design Decisions

2.1 Type in symbol table and type checker

In the design of our AST, we defined an *identifierType* node that could be matched to any possible types. Since type node is the structure we are extensively using for type checking, We simply stored the type node as a type for each symbol in the symbol table.

When comparing two types, we recursively checked each field pointer and stroed informations. If two structs had exactly the same stored informations, they were evaluated as equal.

We also implemented a method for resolving a given type. Given the specified type, we searched the symbol with the same name from the symbol table of current scope to root. Then we recursively called this method on the underlying type stored in the symbol. If the input type was base type, struct type, array type, or slice type, the type itself was returned as the underlying type. For all defined types, if we can't find the symbol with the same name, an undeclared type error would be printed and the program exited with 1.

2.2 Symbol table structure

For implementation of symbol, we constructed a struct that stored the symbol name, declared type, and symbol kind. The symbol kind could be function kind, variable kind, or type kind, which indicated the symbol was for a function declaration, variable declaration, or type declaration.

The symbol table was structured as a cactus stack, with symbols stored inside using a hash map, and a pointer pointing to its parent, and a pointer array pointing to its children. When looking up a symbol, we would traverse through the symbol tables from the current table and all its ancestors until the root table. If a variable was already declared, an error message would be printed and the program exited with 1.

When printing the symbols, we traversed the symbol tables from the root using dfs. At each symbol table, the scope and all the symbols inside were printed. However, at each scope, there was no guarantee on the ording of symbols since symbols were stored using hash map.

3 Detailed Design Decisions for Symbol Table

3.1 Declarations

3.1.1 Variable declarations

The symbol with variable kind was constructed and added into current symbol table when a variable declaration node was met. The variable name and type were stored into the symbol. If a symbol with the same name had already been in the symbol tables, an redeclaration error message was returned. If the type of the variable was not declared in the declaration, it was automatically inferred by our type checker.

3.1.2 Type declarations

Defined types were made as a symbol of type kind, in which the type name and the underlying type were stored. Symbols for base types were created and added in the root symbol table by default before typechecking. If a symbol with the same name had already been in the symbol tables, an redeclaration error message was returned.

3.1.3 Function declarations

We stored function declarations in a symbol with the function name as symbol name and a struct of input type list and output type as symbol type. We also check for special functions "main" and "init" here and ensure that they have neither input nor output requirements. If a symbol with the same name had already been in the symbol tables, an redeclaration error message was returned.

3.2 Statements

3.2.1 Short declaration

When a short declaration statement is encountered, we first search in the symbol table to see whether there are undeclared variables in its identifier list. If not, an error message is printed. (see the later half in 4.3.3)

3.2.2 Op-assignment

3.2.3 Block

We scope the symbol table when encountered a new block and unscope when block is finished.

3.2.4 For, if, and switch statements

For these statements, they might have a simple declaration (i.e. the init statement) before their conditions. We open a child scope in symbol table just to store the variable declaration that may happened in the init statement. By doing so, we both allowed the init statement to redeclare existing variables, and enabled the variable declared in the init statement to be shadowed by declarations that occurred inside the for/if/switch's block body.

4 Detailed Design Decisions for Type Checker

4.1 Declarations

In variable declarations, we have 3 different cases. If the declaration has no expression as value, our compiler put x:T in the symbol table; if the declaration has no declared type but an expression as value, we would infer the type of value; if the declaration has both the declared type and an expression as value, we would infer the type of value, compare it with declared type. For multi-variable declaration we would also be checking the number of variables and number of expressions.

When type checking main and init functions, we made sure that they did't have parameters or return statements.

4.2 Statements

4.2.1 Expression statement

When we encounter an expression statement node, we first check that the it's expression child is a function call, then check whether the function is well typed.

4.2.2 return statement

In order to type check return statements ,we create a global variable called curReturnType, which remembers the return type of the current function. Every time a function declaration is encountered, the curReturnType is updated to the return type of the function. When we traversed the function body and arrived at the return statement, we could compare the the type of the its expression child with the curReturnType to type check the return statement.

4.2.3 Short declaration

For the undeclared variables, we store them into the symbol with the type of the expression they are assigned. For the declared variables, we first search in the symbol table to find out the types of these variables. Then, we compare the types of the variables with the types of the expressions they are assigned. If all declared variables have the same type as the expressions they are assigned, then the short declaration type checks.

4.2.4 Assignment

First, we traverse the identifiers on the left hand side of the assignment statement, search each identifier in the symbol table, and find out their type to make an identifierTypeList. Then, we traverse the expressions on the right hand side of the assignment statement, infer the type of each expression, and make an expressionTypeList. By comparing each entry in the the identifierTypeList with the correspond entry in the expressionTypeList, we type checked the assignment statement.

4.2.5 For and if statements

For the condition expression of a for or if statement, we first resolve its base type, and then check whether this base type is boolean.

4.2.6 switch statements

If the switch statement doesn't have an expression, we resolve the expressions of its switch cases and check whether they have the base type of boolean.

4.2.7 op-assignment

For op-assignment, the left hand side was a primary expression, we use inference to get its type, then check if both sides were equal and if they followed the operator's typing rule.

4.2.8 Increment/decrement statements

We first resolve the type of the operand to find its base type, then check whether the base type is an numeric type.

4.2.9 Print/println statement

If the expression child of the print statement is not NULL, we resolve each expression to find its base type, and check whether its base type belongs to int, float64, boolean, rune, or string.

4.3 Expressions

We built a function called inferTypeExp and some helper functions to infer and typecheck expressions. The function inferTypeExp would continue traversing the AST in dfs manner, typecheck each expression given the returned type of constituting expressions, and decide a type for the expression following the rules given in the specifications.

4.3.1 Literals and Identifiers

The types of literals were inferred by their node kinds in AST. Literals were stored as different kinds corresponding to their types based on our design of AST. The types of identifiers were inferred by querying the symbol table so that the type of first found symbol from current symbol table to root was returned.

4.3.2 Unary exression

We typechecked unary expressions by checking if the constituting expression can be resolved to specified types in specifications. If successful, the unary expression would be assigned the

same type as the type of constituting expression. Otherwise, it didn't typecheck and an error was returned.

4.3.3 Binary exression

We first checked if the two constituting expressions were typechecked and had the same inferred type returned. Then, if the inferred type could be resolved to the specified types, the binary expression was well-typed and assigned the same inferred type or bool in comparison expressions.

4.3.4 Builtin functions

When reached builtin expression nodes in AST, we checked if the expression was well-typed and could be resolved to []T or [N]T (or string type for len(expr)). For append builtin, we also checked the second expression was well-typed and resolved to T, then the type of first expression was assigned and returned. For the rest 2 builtins, type int was assinged and returned.

4.3.5 Field selection and Indexing

They were typechecked strictly followed the rules in specifications.

4.3.6 Function call and Type casting

Type cast would be parsed as a function call in the parser. However, we can separate the two by querying the symbol table, and see matched symbol was a type kind symbol. If so, then proceed to typecheck following the rules for type cast. Otherwise, we typechecked it following the rules for function in specifications.

5 Miscellaneous Topics

5.1 Scoping Rules

1. The package name, import statement, and top-level declarations all belongs to the initial scope. 2. open a new child scope every time an opening curly parenthesis is encountered, and return to parent scope when the closing curly parenthesis is encountered. 3. function's return type belongs to the current scope, while function parameters belong to the child scope. 4. The if, for, and switch statement as a whole belong to current scope, their init statements belongs to the child scope, and the content inside their curly parenthesis belongs to the grandchild scope.

5.2 Invalid Program Rules

Program number and rules in Milestone2 specifications for invalid program were listed below.

1. Rule 3.10: The program is invalid because the expression of the print statement is not well typed since the expression is an identifier that has not been declared yet.
2. Rule 2.1: The program is invalid because the variable x has already been declared in current scope.

3. Rule 2.2: The program is invalid because the type has already been declared in current scope.
4. Rule 2.1: The program is invalid because 2 function parameters can't have the same name.
5. Rule 2.4: The program is invalid because main function can't have return type.
6. Rule 3.4: The program is invalid because the function has a return type but the return statement doesn't have an expression.
7. Rule 2.1: The program is invalid because the identifier doesn't have the same type as the expression that it is assigned.
8. Rule 2.3: The program is invalid because the function f has already been declared in current scope.
9. Rule 3.14: The program is invalid because the expression of the increment statement doesn't resolve to a numeric base type.
10. Rule 4.5: The program is invalid because the expression of the function call doesn't have the same type as the function parameter.
11. Rule 4.6: The program is invalid because the index of the array doesn't resolve to int.
12. Rule 4.7: The program is invalid because the variable that the field selector is apply to is not of type struct.
13. Rule 4.7: The program is invalid because struct doesn't have the field being selected.
14. Rule 4.3: The program is invalid because the expression of the logical negation doesn't resolve to a bool.
15. Rule 2.4: The program is invalid because init function can't have return type.
16. Rule 3.4: The program is invalid because the return statement's expression's type doesn't match the return type of the function.
17. Rule 3.10: The program is invalid because the expression of the print statement is not well typed since the expression is an identifier that has not been declared in current scope or parent scope.
18. Rule 3.13: The program is invalid because the case's expression doesn't have the same type as the switch statement's expression.
19. Rule 3.12: The program is invalid because the expression in the for condition doesn't resovlve to type bool.
20. Rule 2.1: The program is invalid because the identifier doesn't have the same type as the expression that it is assigned since the return type of the function call doesn't have the same type as the identifier.
21. Rule 4.4. The lhs and rhs of a binary expression doesn't have the same type, so the binary expression doesn't type check.
22. Rule 4.5: The special function init may not be called as a function call.
23. Rule 4.6 Indexing expr[index]. The program is invalid because index resolves to float64 and doesn't resolve to type int.

24. Rule 4.7 Field selection: expr.id. The program is invalid because S doesn't have a field named id.
25. Rule 4.7 Field selection: expr.id. The program is invalid because S doesn't resolve to a struct type.
26. Rule 4.8.1 Append: append(e1, e2). The program is invalid because S doesn't resolve to a []T.
27. Rule 4.8.2 Capacity: cap(expr). The program is invalid because S doesn't resolve to a []T or [N]T.
28. Rule 4.9 Type cast. The program is invalid because type cast can't take more than one expression argument.
29. Rule 4.9 Type cast in specifications. A type cast expression is well-typed if: type resolves to a base type int, float64, bool, rune or string.
30. Rule 4.9 Type cast in specifications. The program is invalid because expr doesn't have a type that can be cast to the specified type.

6 Contribution

Yue Lyu: Compose report, implement type checker for declaration and part of symbol table. Chunyao Zhao: Compose report, implement type checker for statement and part of symbol table. Yiyang Liu: Compose report, implement type checker for expression and part of symbol table.