GoLiteTypechecker

Akshay Gopalakrishnan and Stuart Mashall

March 2019

1 Design Choices

1.1 Parsing Issues

1. The blank identifier was to be dealt with in the parsing phase. But instead we decided to not detect blank identifiers in the parsing phase and have them specially tagged in the typechecker phase.
2. Optional statements in the "for", "if" and "switch" clauses were replaced with definite statements. To accomodate the optional ones, we added a new type of statement as a variant called Empty Statement and generated the AST for that accordingly
3. We decided to keep type cast expression as part of the function call grammar and chose to deal with type cast separately in the typechecker phase.
4. We dealt with the inability of our grammar to incorporate multidimensional array and slices by modifying our grammar and refactoring our AST
5. We made the indexing values for initialization of arrays to only take literal int (previously we had decided to keep it general and deal with it in the typechecker phase)

1.2 Typechecker Decisions

1.2.1 The Generic Types "Golitetypes"

1. We decided to keep a variant of type "gltype" that has all the type templates we need for the symbol table
2. The variant contains template for the:
   (a) Basic Types - No specific type decalred so it resorts to default type in variants
   (b) Defined Types - Has type (string)
   (c) Struct - A list of tuples of the form (string list * gltype)

   (d) Array - Has type (int * gltype)
   (e) Slice - Has type (gltype)
   (f) Function - Has type of the form (gltype list * gltype) —- argument types * return type

3. We used this gltype to define our last column of the symbol table. (as mentioned in the milestone specs)

1.2.2 The Cactus Stack

1. We decided to keep our cactus stack nodes (called as csnode) as a variant which has two types viz .

   (a) CsRoot - Has the default type for variants
   (b) CsNode - Has a type of literal record (a feature in ocaml from version 4.03) THe record has :
      i. parent - Whcih is of type (csnode)
      ii. children - To store all the children scopes which is of type (ref csnode list) —— The type csnode will be inferred on the first entry and it is of ref type as we will be using it to assign to our current scope
      iii. context - Which stores the actual contents of the symbol table which is of type (string (category, gltype)) Hashtable ——— The string , category and gltype are the 3 columns of our symbol table as specified in the milestone specs

   We decided to keep CsRoot to detect when we are at the root scope above all other scopes

2. We decided to keep 2 scopes at the global level which are root scope and global scope

   (a) Root scope has all the default entries in the symbol table as defined in the milestone specs
   (b) Global Scope is a child of the root scope. This scope will be used for whatever is defined in the global context

3. We defined a variable called "current_scope" of ref type csnode which initially points to the global scope. This variable will be changed as we shift scopes.

1.3 Implementation

1.3.1 Helper Functions

1. We created several helper functions to assist us in the typechecking part. Most of it was related to fetching or finding the existence of symbols in the cactus stack. Some of the main ones are as follows:

   (a) get_vcat_gltype_from_str – Fetches the category and gltype as a tuple on searching given string on the symbol table hash map
   (b) err_if_id_in_current_scope – If given string is in current scope raise an error
   (c) err_if_id_not_declared – A recursive function to check if a particular string is not declared in the cactus stack
   (d) err_if_type_not_declared – A recursive function to check if the given type (as a string) is not there on the cactus stack
   (e) rt – A recursive function to resolve a type to one of its basic root type (root meaning the first definition of that type)
   (f) create_new_scope – Creates a new scope and sets the current scope to that scope
   (g) get_parent_scope – Sets the current scope to the parent of the current scope (beware of the confusion XD)

2. We used the above as well as a few more helpers to structure our whole typechecking. We first started by creating a function for each of the unique variants in the AST that we defined. For each variant we called / used one or more of the above helpers to do the typechecking.

3. As an important note, we relied heavily on using List operations that involve fold, map and iter functionalities in conjunction with the above helpers to get our job done.

1.3.2 Problems

1. The first problem we tackled was that of variable declarations :
   (a) Since our AST defined this section as a list of variable declarations, it seemed natural to use List.iter to typecheck each of them
   (b) We defined a method type_check_vd to actually check each variable declaration
   (c) This function entailed using the helper functions to adhere to checks and adding the details to the symbol table
   (d) In order to tackle a simultaneous list of variables declared at the same time , we had to check the existence of expressions at the right side. This we handled by using a fold operation using List.fold_left that would return true if all expressions on the RHS of variable declarations existed.

2. The next was on type declarations :
   (a) Similar to variable declarations we used List.iter to typecheck each of them
   (b) We defined a method type_check_td to do this work for us

   (c) This function too used the helper functions to adhere to the specs of golite and add the contents to the symbol table
   (d) For type declaration of struct, we came to an issue of returning the gltype of the struct. We then, decided to use List.map to return types of every declaration inside along with their names, which let us return a composite gltype of the struct. (This took quite some time)

3. We tackled function declarations at a later phase. The only problem we faced was that of return check. Here is how we resolved it
   (a) We realized that the last statement in the statement block of a function should be return. If not then the typechecker should most porbably fail.
   (b) If the last statement is not return, we check what does the last statement return. For this we forced our statement typecheck function to always return a gltype (Void type for default)
   (c) Doing this, we check if the return type of the last statement does match that of the function, if so we pass.
   (d) On the way we also check the return types within "if", "for" and "switch" clauses. Anytime a mismatch, we record it and check it appropriately. This resloved all our return statement issues

4. Next we handled the typechekcing of expressions as it was required for variable expressions
   (a) List.iter was used to handle expression lists
   (b) The type check of expressions was fairly strightforward as we had our helper functions ready for them
   (c) We created another function called pt_if_rt which checks if resolved type of identifier that we give belongs to a type in a list that we give (This list is defined in the set of helper functions)
   (d) For type equality we created another function pt_if_type_check_eq just checks if two expressions have the same type

5. Next we handled statements as it was the final bit of the typechecker
   (a) The only place we felt a little hard was the short hand declarations.
   (b) FOr this, we created tuples using zip function and we created a list of these tuples
   (c) We then wrote a function that does the at-least-one check followed by type check equality / adding the variable onto table with appropriate type
   (d) Finally we sent this function to List.iter to iterate over all the tuples

1.4 Last Statements

1. We worked on the entire typechecker phase together. So there is nothing as division of work.
2. There are around 110 test cases in total which is there online with the code itself.
3. The code was tested exhaustively with our test cases
4. Some invalid cases do pass, but we have ran out of time to resolve them
5. Please note that this code works for ocaml version beyond 4.02 (ie 4.03 and above)