Milestone 2

Implementation

Type system

The implementation is rather simple. It's a pretty straight-forward translation of the Golite types to Java classes.

They all implement the `Type` class, which has abstract methods for classification (numeric, integer, comparable and ordered), and an abstract method for the resolve operation. The `equals()` and `toString()` methods are also implemented in each type. The latter prints types in the Go format; useful for debug and error messages.

The basic types are singletons in the `BasicType` class. Aside from implementing the `Type` methods, they also provide a few more for casting rules.

The `IndexableType` is the parent of the `ArrayType` and `SliceType`. It simplifies type-checking for operations performed on either kind. It also stores the common component type. Otherwise, the types just implement the abstract methods.

The `StructType` class is just an implementation of `Type`. It stores the fields as a list of `StructType.Field` objects, which store the type and name of each field.

The `FunctionType` is just an implementation of `Type`. It stores the function signature information, which is the parameters and return type. It's used as the type of identifier expressions that reference functions, which is necessary since Go directly calls values.

Finally the `AliasType` is a name and a reference to any other type. It's the only one that doesn't return itself in the implementation of resolve. Instead it unwraps the aliased type of any further aliases. This type also needs the ID of the context in which it was declared, with the name of the alias. This is because aliases are only unique within different scopes. If two alias objects are created for the same symbol, then they should be equal because it is the same name and scope. Relying on the `==` operator would always return false, but just comparing names would return true for aliases in different scopes. So we add the context ID to properly implement the equality method.

Symbols

All the symbols implement the `Symbol` class. They all have a name and a type. There is also some associated source position information to provide better error messages.

The first kind of symbol is the `DeclaredType`. This class doesn't add anything to the parent. It's used for type declarations. While type declaration are normally always aliases, this isn't the case for the pre-declared basic types in the universe context. For this reason, the class will accept any kind of type. When an alias type is used, it ignores the alias and actually uses the type being aliased. This might seem redundant, but it necessary to support the special case of the pre-declared types.

The `Variable` symbol also doesn't add anything to the parent. Here we don't care about the value, so we only need the name and the type. There is also a `constant` flag, which was added to prevent pre-declared variables in the universe block from being redeclared, but it's unused.

The `Function` symbol is a function type and the name of the function. Again, we don't actually care about the body, since it's not necessary to type-check usage.

Context system

The `Context` is basically the symbol table. It stores a map of names to `Symbol`s. Furthermore, every context is assigned a serial ID, which is used for uniqueness and debugging. Every context, except the universe, has a parent. It can be recursively searched when looking up a symbol. This implement the block shadowing rule. We can also declare a symbol in a context. This simply adds it to the map, checking for duplicates. If the identifier is blank, then nothing is added (as per the Go specification). For checking return statements, we need the type of the enclosing function. We added a method for this purpose, which traverses the context hierarchy until a `FunctionContext` is found. The remaining methods are used to print and convert contexts to strings, for debugging and the `-dumpsymtab` option.

The top-most context is the `UniverseContext`. It's the only one that doesn't have a parent. It also always has the ID 0. It is used as a singleton. The pre-declared variables and identifiers are always present, and the `declareSymbol()` method is overridden to always throw an exception.

The next context is the `TopLevelContext`. Its parent is always the

`UniverseContext`. Its ID is also always 1. Otherwise there isn't anything special about it.

The next context is the `FunctionContext`. Its parent is always the `TopLevelContext`. It doesn't allow declaration of nested functions. It also stores the `Function` symbol for the declaration. When searching for the enclosing function, it returns that symbol.

The last kind of context is the `CodeBlockContext`. Its parent is either a `FunctionContext` or another `CodeBlockContext`. It's used for everything else: blocks; "if-else", "for" and "switch" statements; and switch cases. This data is stored for better debug information.

Type checker

Declarations

Type declarations simply create a new `DeclaredType` symbol in the context. The type is an alias of the declared one, using the name of the declaration.

Variable declarations are simply an implementation of the specification. When no type is given, the value type is used. Otherwise, it must be the same as the explicit type. The variables are declared in the context, which will throw an exception if the name is already used.

Function declarations open a new `FunctionContext`, then declares the parameters as variables in this context. After type-checking the body, it is checked for a terminating statement at the end. This is done with a separate `AnalysisAdapter` that implements the specification.

Short variable declarations will result in at least one new identifier being declared in the current context. The type-checker ensures that at least one identifier on the left-hand side has not been declared by looking up the identifier in the current context. If this condition is met and all of the expressions on the right-hand side are well-typed, then all previously undeclared identifiers are declared in the current context.

Statements

Empty statement, `continue` and `break` are trivially well-typed, and are handled by overriding the standard visitor methods in `AnalysisAdapter` an empty method.

Return statements with an expression result in the expression being evaluated and its type being compared with that of the enclosing function in the context. If the types are the same then the return statement is well-typed. If no expression is given, a check is done to ensure that no return type is given for the enclosing function.

Assign statements are first dealt with by ensuring that all the identifiers on the left-hand side have been declared (or are the blank identifier) and type-checking the right-hand side to ensure that all given expressions are valid. The identifier and expressions lists are then traversed to ensure that the types of each identifier-expression pair are the same.

The `print` and `println` statements result in a type-check of all given expressions to ensure that any `AliasTypes` resolve to a `BasicType`. If no expressions are given, the `print` and `println` statements are trivially well-typed.

Declaration and short-declaration statements are type-checked using the

method described in the 'Declarations' section.

For-loop statements result in the creation of a new `CodeBlockContext` for the loop condition. If a for condition is given, the expression is type-checked to ensure that its type resolves to 'bool.' A new `CodeBlockContext` is opened for the body of the loop, and all of its statements are type-checked. The post-condition (if given) is type-checked after the statements in the loop body have been type-checked.

If-statements first type-check the if-block by doing the following:
A new `CodeBlockContext` is created for the `init` statement (if given) and the expression used as the if-condition. The statement and expression are both type-checked and a check is done to ensure that the expression type resolves to bool. Then a new `CodeBlockContext` is created for the if-body and all of the statements in the body are type-checked.

To keep track of the scope of all statements inside the switch block, a new `CodeBlockContext` is used for the body of the switch statement. Thereafter, the `init` condition is type-checked. Then, the type for follow up expression is evaluated and stored in a variable to make sure it matches with the type of individual case conditionals. If the expression is not specified, the type of the expression is assumed to be `bool`. Thereafter, each case conditional type is evaluated and compared with the expression's type. Thereafter, all statements inside each case block are type-checked. For default case, no extra type-checking is needed. A new `CodeBlockContext` is used to keep track the scope of all statements inside each case block.

For op-assignments, as the left and right hand sides of each statement evaluates to an expression, we re-use the same method that we use to

type-check expressions. The comments above about resolving expressions apply here as well. In a nutshell, the method ensures that all the expressions on the left and right hand sides of each statement resolve to valid type for for the operation, are well typed and that the type of left hand side is equal to the type of right hand side.

Expressions

The literal expressions simply return their associated type.

For identifier, we look up the symbol. It must exist and be either a variable or function. Otherwise and exception is thrown. Functions are treated a callable values, but since no other operation supports function types, they can't be used for anything else.

Select expressions check if the value is a struct type and has a field of the selected name. If it does then the field type is returned. Otherwise an exception is thrown.

Index expressions check that the value is an array or slice (`IndexableType`) and that the index is an `int`.

Calls have a special case for casts. If the value being called is an identifier expression and the identifier references a type symbol, then we type-check it as a cast instead. Otherwise we check that the value has a function type, and that it has a return type. If the arguments are assignable to the parameters, then the expression has the same type as the return type.

For casts, we check that both the symbol and argument (which there should only be one of) resolve to basic types, and that they are valid for casting (we check casting from and to). The type is then the same as the

casting type.

For appends, the first argument must be a slice and the second must be the same as the component type.

For unary and binary operators, we check that the resolved type(s) are of the proper category (integer, numeric, etc.) for the operator. Additionally, for binary types, we have to check that the unresolved types are the same. The resulting type depends on the operator.

Types

Named types just return the type of the `DeclaredType` symbol they reference.

For slice types, we wrap the component type in `SliceType`. Same for arrays, but we also include the length, which we obtain by evaluating the integer literal.

For struct types, we create the fields first, then wrap them in a `StructType`.

Team work summary

Aleksi Sapon wrote the type, symbol and context systems. He also implemented the type-checking for declarations (except for short variable declarations), expressions and types. He also implemented the terminating statement checker, and added the printing capabilities to the application. He wrote seven invalid type test cases. Finally he also took care of overall code quality and consistency.

Rohit Verma implemented the type-checking for statements including the if-else statement, the switch statement, the print statements and all the op-statements and made appropriate changes to the other parts of code as and where appropriate. He also thoroughly tested the code for any incorrect or missing cases in the type-cheker. Furthermore, he wrote seven invalid type test cases.

Ayesha Krishnamurthy also worked on type-checking for statements, including short declarations, assignments, for loops, if statements, expressions and print statements. Aside from her work with statements, she contributed to testing and bug fixes and wrote nine invalid type test-cases.