Design Document for Milestone 2

Lore, J., Lougheed D., Wang A.

March 17, 2019

Contents

This document is for explaining the design decisions we had to make whilst implementing the components for milestone 2.

1 Weeding

We implemented additional weeding passes for certain constraints that could be verified either at the weeding level or the typecheck level, since in most cases it was easier to check via weeding. The constraints we use weeding to check for this milestone are:

- Checking correct use of the blank identifier. It was much easier to recurse through the AST and gather all identifiers that cannot be blank, and then check this whole list, versus checking usage in the type-checking pass. Additionally, because we have offsets in our AST, we could easily point to the offending blank identifier without having to make error messages for each specific incorrect usage, as it is obvious what the incorrect usage is when we print out the location of the blank identifier.
- Ensuring non-void function bodies end in return statements. It was easier to do a single weeding pass; otherwise, we'd have to recurse differently on functions that have a return type versus functions that don't have a return type during typechecking. Essentially, we'd need context-sensitive statement typechecking, with two different versions depending on return type. This was not deemed worth it compared to a single weeding pass.
- Ensuring `init` function declarations do not have any non-void return statements. Similar to the above, it is easier to do a weeding pass then require context-sensitive (i.e multiple different) traversal functions given the context of the current function declaration.
- Ensuring any 'main' or 'init' function declarations do not have any parameters or a return type. This is again a straightforward weeding check. It could also have been done via typecheck, but it is slightly easier to implement with weeding passes (no symbol table required).
- Checking that any top-level declarations with the identifier `init` or `main` are functions, since they are special identifiers in Go (and GoLite) in the global scope.

2 Symbol Table

In Haskell, data structures are typically immutable, and much of the language is designed around this. One of the main design decisions made around the symbol table was deciding whether to go with an immutable or mutable symbol table. In an immutable symbol table, a new symbol table would have to be made every time a scope is added or modified. Right away, despite this being a conceptually better fit for the language, the potential performance degradation of constantly re-building the symbol table becomes evident.

As a result of this performance impact, we decided on using a mutable symbol table, with mutability supported via Haskell's ST monad. The constraint provides runST (which removes the `ST`, i.e. mutability, from an interior data structure). This is proven to keep functions pure (see A Logical Relation for Monadic Encapsulation of State by Amin Timany et al.). This made the `ST` monad a better choice than other monads providing mutability (the main example being `IO`, which if used would have resulted in all our functions after symbol table generation being bound by `IO`, i.e. impure and also harder to work with, as they are wrapped by an unnecessary monad). The trade-off of this decision was a large increase in the difficulty of implementing the symbol table, which made up a huge portion of the work for this milestone. However, once we finish using the symbol table, our final result (a typechecked/simplified, proven-correct AST) is pure and very easy to manipulate for the `codegen` phase in the next milestone.

For symbol table storage, we created a new data type `Symbol` to represent each symbol, analogous to the different type of symbols (types, constants, functions, and variables). A `Symbol` can be one of:

- `Base`: Base types (`int`, `float64`, etc.)
- `Constant`: Constant values (only used for booleans in GoLite)

- `Func [Param] (Maybe SType)`: Function types, with parameters and an optional return type.
- `Variable SType`: Declared variables, of type SType.
- `SType SType`: Declared types (note: in Haskell, the first `SType` here is a constructor, and the second `SType` is a data type attached to the constructor.) In this way, all possible symbol types are encompassed by a single Haskell data type.

We also created a new data type `SType`, which is used to store vital information concerning the types we define in the original `AST`. An `SType` can be one of:

- `Array Int SType`: An array with a length and a type.
- `Slice SType`: A slice with a type.
- `Struct [Field]`: A struct with a list of fields.
- `TypeMap SIdent SType`: A user-defined type, with an identifier and an underlying type, which may be recursively mapped, eventually to a base type.
- `PInt, PFloat64, PBool, PRune, PString`: Base types.
- `Infer, Void`: Special ~SType~s for inferred types and void return values.

In this way, all possible GoLite types, including user-defined types, are accounted for.

All types that are left to be inferred during symbol table generation are updated when typechecking (we infer the type of variables and then update their value in the symbol table, to make sure things like assignments don't conflict with the original inferred type).

2.1 Scoping Rules

The scoping rules we used/considered are as follows:

- Function declarations: the parameters and function body are put in a new scope, but the function itself is declared at the current scope. Note that here we had to treat the function body as a list of statements and not a block statement, because if we recursed on the block statement our block statement rule would put the function body in a new scope. Instead, the body must be in the same scope as the parameters.
- Block statements are put into a new scope.
- If statements: we open a new scope, containing the simple statement and expression condition at the top level, and then other scope(s) inside for the body/bodies: one for `if`, and one for `else` (if there is one). If an `else` is present, the if and else scope are siblings.
- Switch statements: open a new scope for the `switch`, and another scope for each switch `case`. All switch case scopes are siblings.
- For loop: open a new scope, with optional clauses put at the top level (simple statements 1 and 2, and condition). The body is put in a nested scope.

3 Type Checker

For type-checking, we decided on a single-pass approach, combining combined symbol table generation and statement type-checking. This improves performance, and is feasible as a product of GoLite's declaration rules, which specify that identifiers must be declared before they can be used.

The other approach we considered involved generation of a type-annotated AST, with types of expressions contained in the AST, so that we could get rid of `ST` mutability from the symbol table as soon as possible (some of us did a similar thing for the assignment, but this was mainly relevant for print statements in C codegen needing to know the type of the expression they're printing).

We decided on doing all typechecking at the same time as symbol table generation because type inference has to be done to generate this new AST, and type inference requires typechecking (e.g. `"a" + 5` has no inferred type, since the expression is undefined; we only know this because of typechecking).

At first, we were going to generate an annotated AST only to typecheck things that aren't expressions. However, at that point, since we were already doing one in-depth pass of the original AST for symbol table generation, we decided that we might as well do the other half of typechecking in the same phase, since it seemed odd to split typechecking between the symbol table and a separate pass. The alternate approach may have been more feasible if type inference did not require typechecking, but in GoLite it did not seem to make sense. Therefore, after the one pass of our original AST, the final result is a typechecked AST, with extremely limited type annotation.

Additionally, we decided to resolve all type mappings (except for structs) to their base types when generating this new AST: all the casts/equality checks/new type usages are already validated in typechecking, so we don't need them anymore, nor do we need the mappings. Thus, our new AST was also able to get rid of type declarations (except for structs).

4 New AST

As mentioned above, dependency on the SymbolTable results in a dependency on the `ST` monad, which adds complexity to each operation. As a result, our goal after typechecking is to create a new AST, which reflects the new constraints we enforce. Namely:

- Typecheck errors are caught beforehand, so we no longer need offsets, or error breakpoints.
- All variables are properly typechecked, and can therefore reference an explicit type. Each type is composed of parent types up until the primitives. This includes cases like function signatures, where we can associate each parameter with a type instead of allowing lists of identifiers to map to a single type. In preparation for codegen, we can then use our new AST exclusively, without any other mutable data structures. Any additinoal information we need can be added back into the AST, with minimal changes to models used at previous stages.

5 Invalid Programs

Summary of the check in each invalid program:

- `append-diff-type.go`: Append an expression of a different type than the type of the expressions of the `slice`.
- `append-no-slice.go`: Append to something that isn't a slice.
- `assign-no-decl.go`: Assign to a variable that hasn't been declared.
- `assign-non-addressable.go`: Assign to a LHS that is a non-addressable field.
- `cast-not-base.go`: Cast to a type that isn't a base type.
- `dec-non-lval.go`: Decrement something that isn't an `lvalue`.
- `decl-type-mismatch.go`: Declare and assign variable of explicit type to an expression of a different type.
- `float-to-string.go`: Try to cast a `float` to a `string`.
- `for-no-bool.go`: While variant of for loop with a condition that isn't a bool.

- `func-call.go`: Function call with arguments of different type than function declaration arguments.
- `func-no-decl.go`: Calling a function that hasn't been declared.
- `function-already-declared.go`: Trying to declare a function that has already been declared.
- `function-duplicate-param.go`: Trying to declare function with two params with same name.
- `if-bad-init.go`: If with an init statement that does not typecheck (assignment of different type).
- `inc-non-numeric.go`: Increment an expression that doesn't resolve to a numeric base type.
- `index-not-list.go`: Index into something that isn't a slice.
- `index.go`: Index that does not resolve to an int.
- `invalid-type-decl.go`: Declare a type mapping to a type that doesn't exist.
- `no-field.go`: Using selector operator on struct that doesn't have the field requested.
- `non-existent-assign.go`: Assigning a variable to a non existent variable.
- `non-existent-decl.go`: Trying to declare a variable of a type that doesn't exist.
- `op-assign.go`: Op-assignment where variable and expression are not compatible with operator (i.e. `int + string`)
- `print-non-base.go`: Trying to print a non base type.
- `return-expr.go`: Returning an expression of different type than the return type of the function.
- `return.go`: Return nothing from non-void function.
- `short-decl-all-decl.go`: Short declaration where all variables on LHS are already declared.
- `short-decl-diff-type.go`: Short declaration where already defined variables on LHS are not the same type as assigned expression.
- `switch-diff-type.go`: Type of expression of case is different from switch expression type.
- `type-already-declared.go`: Trying to define a type mapping to a type that already exists.
- `var-already-declared.go`: Trying to declare a variable that is already declared.

6 Team

6.1 Team Organization

The three main components for this milestone are the symbol table, type checking rules, and new AST; as well as tests for all three. Development of these components was lead by Julian, David, and Allan respectively. As there is a high degree of coupling between each component, we continually sought feedback from one another. The component leads are in charge of understanding the overall component and in resolving concerns or requests from other members.

6.2 Contributions

- **Julian Lore:** Implemented weeding of blank identifiers, symbol table generation, typecheck (aside from type inference and expression typechecking) and submitted invalid programs.
- **David Lougheed:** Worked on expression type-checking and type inference, including tests. Also worked on the weeding pass for return statements.
- **Allan Wang:** Added data structures for error messages, and supported explicit error checking in tests. Created the data model for symbol table core. Added hspec tests.