COMP520 - GoLite Type Checking Specification

Vincent Foley

February 26, 2015

1 Declarations

Declarations are the primary means of introducing new identifiers in the symbol table. In Go, top-level declarations can come in any order; in GoLite, we will require that identifiers be declared before they are used. This will prevent mututally recursive functions, however it should make the type checker implementation easier.

The symbol table should start with a few pre-declared mappings. These mappings may be shadowed.

Identifier	Type
true	bool
false	bool

1.1 Variable declarations

```
var x int
```

Adds the mapping `x -> int` to the symbol table.

```
var x int = expr
```

If `expr` is well-typed and its type is `int`, the mapping `x -> int` is added to the symbol table.

```
var x = expr
```

If `expr` is well-typed and its type is `T`, the mapping `x -> T` is added to the symbol table.

In all three cases, if `x` is already declared in the current scope, an error is raised. If `x` is already declared, but in an outer scope, the new `x -> T` mapping will *shadow* the previous mapping.

Note: In Go, it is an error to declare a local variable and not use it. In GoLite, we will allow unused variables. (If you wanted to comply with the Go specification, how would you make sure that all locals are used?)

1.2 Type declarations

```
type num int
```

Adds the type mapping `num -> int` to the symbol table. If `num` is already declared in the current scope, an error is raised. If `num` is already declared, but in an outer scope, the new `num -> int` type mapping will *shadow* the previous mapping.

1.3 Function declarations

```
func f(p1 T1, p2 T2, ..., pn Tn) Tr {
        // statements
}
```

Given the declaration for `f` above, the mapping `f -> (T1 * T2 * ... * Tn -> Tr)` is added to the symbol table. If `f` is already declared in the current scope (i.e. the global scope since we don't have nested functions), an error is raised.

For each formal parameter `pi`, the mapping `pi -> Ti` is added to the symbol table. If two parameters have the same name, an error is raised. A formal parameter may have the same name as the function itself.

A formal parameter or a variable or type declared in the body of the function may have the same name as the function.

```
// Valid
func f(f int) {
        ...
}

// Invalid
func f(f int) {
        var f float64  // Redeclares f (the formal parameter)
        ...
}
```

A function declaration type checks if the statements of its body type check. Additionally, for functions that return a value, there should be a well-typed return statement on every execution path.

Hint: you may want to add a new weeding pass to check for return statements on every path.

2 Statements

Type checking of a statement involves making sure that all its children are well-typed. A statement does **not** have a type.

2.1 Empty statement

The empty statement is trivially well-typed.

2.2 `break` and `continue`

The `break` and `continue` statements are trivially well-typed.

2.3 Expression statement

`expr`

An expression statement is well-typed if its expression child is well-typed.

2.4 `return`

`return`

A `return` statement with no expression is well-typed if the enclosing function has no return type.

`return expr` A `return` statement with an expression is well-typed if its expression is well-typed and the type of this expression is the same as the return type of the enclosing function.

2.5 Short declaration

`v1, v2, ..., vn := e1, e2, ..., en`

A short declaration type checks if:

- All the expressions on the right-hand side are well-typed;
- At least one variable on the left-hand side is not declared in the current scope;
- The variables already declared in the current scope are assigned expressions of the same type. E.g. if the symbol table contains the mapping `v1 -> T1`, then it must be the case that `typeof(e1) = T1`.

If these conditions are met, the mappings `v1 -> typeof(e1)`, `v2 -> typeof(e2)`, ..., `vn -> typeof(en)` are added to symbol table.

2.6 Declarations

Declaration statements obey the rules described in the previous section.

2.7 Assignment

`v1, v2, ..., vn = e1, e2, ..., en`

An assigment statement type checks if:

- All the expressions on the left-hand side are well-typed;

- All the expressions on the right-hand side are well-typed;
- For every pair of lvalue/expression, `typeof(vi) = typeof(ei)`.

2.8 Op-assignment

```
v op= expr
```

An op-assignment statement type checks if:

- The expression on the left-hand side is well-typed;
- The expression on the right-hand side is well-typed;
- The operator `op` accepts two arguments of types `typeof(v)` and `typeof(expr)` and return a value of type `typeof(v)`.

2.9 Block

```
{
        // statements
}
```

A block type checks if its statements type check. A block opens a new scope in the symbol table.

2.10 `print` and `println`

```
print(expr)
println(expr1, expr2)
```

A print statement type checks if all its expressions are well-typed.

2.11 For loop

```
for {
        // statements
}
```

An infinite for loop type checks if its body type checks. The body opens a new scope in the symbol table.

```
for expr {
        // statements
}
```

A "while" loop type checks if:

- Its expression is well-typed and has type `bool`;

- The statements type check.

The body opens a new scope in the symbol table.

```
for init; expr; post {
        // statements
}
```

A three-part for loop type checks if:

- `init` type check;
- `expr` is well-typed and has type `bool`;
- `post` type checks;
- the statements type check.

The `init` statement can shadow variables declared in the same scope as the `for` statement. The body opens a new scope in the symbol table and can redeclare variables declared in the `init` statement.

2.12 If statement

```
if init; expr {
        // then statements
} else {
        // else statements
}
```

An if statement type checks if:

- `init` type checks;
- `expr` is well-typed and has type `bool`;
- The statements in the first block type check;
- The statements in the second block type check.

The `init` statement can shadow variables declared in the same scope as the `for` statement. The bodies both open a new scope in the symbol table and can redeclare variables declared in the `init` statement.

2.13 Switch statement

```
switch init; expr {
case e1, e2, ..., en:
        // statements
default:
        // statements
}
```

A switch statement with an expression type checks if:

- `init` type checks;
- `expr` is well-typed;
- The expressions `e1`, `e2`, `...`, `en` are well-typed and have the same type as `expr`;
- The statements under the different alternatives type check.

```
switch init; {
case e1, e2, ..., en:
        // statements
default:
        // statements
}
```

A switch statement without an expression type checks if:

- `init` type checks;
- The expressions `e1`, `e2`, `...`, `en` are well-typed and have type `bool`;
- The statements under the different alternatives type check.

3 Expressions

Type checking of an expression involves making sure that all its children are well-typed **and also** giving a type to the expression itself. This type can should be stored (either in the AST itself or in an auxiliary data structure) as it will be queried by the expression's parent.

3.1 Literals

```
42                // int
1.62              // float64
'X'               // rune
"comp520"         // string
```

The different literals have obvious types:

- Integer literals have type `int`
- Float literals have type `float64`
- Rune literals have type `rune`
- String literals have type `string`

3.2 Identifiers

```
sum
```

The type of an identifier is obtained by querying the symbol table. If the identifier cannot be found in the symbol table, an error is raised.

3.3 Unary exression

```
unop expr
```

A unary expression is well-typed if its sub-expression is well-typed and has the appropriate type for the operation. In GoLite, the type of a unary expression is always the same as its child.

- Unary plus: `expr` must be numeric (int, float64, rune)
- Arithmetic negation: `expr` must be numeric (int, float64, rune)
- Logical negation: `expr` must be a bool
- Bitwise negation: `expr` must be an integer (int, rune)

3.4 Binary expressions

```
expr binop expr
```

A binary expression is well-typed if its sub-expressions are well-typed, are of the same type and have appropriate types for the operation. The type of the binary operation is detailed in the table below. The Go specification (links below) explains which types are ordered, comparable, numeric, integer, etc.

<table>
<tr><th>arg1</th><th>op</th><th>arg2</th><th>result</th></tr>
<tr><td>bool</td><td>||</td><td>bool</td><td>bool</td></tr>
<tr><td>bool</td><td>&&</td><td>bool</td><td>bool</td></tr>
<tr><td>comparable</td><td>==</td><td>comparable</td><td>bool</td></tr>
<tr><td>comparable</td><td>!=</td><td>comparable</td><td>bool</td></tr>
<tr><td>ordered</td><td><</td><td>ordered</td><td>bool</td></tr>
<tr><td>ordered</td><td><=</td><td>ordered</td><td>bool</td></tr>
<tr><td>ordered</td><td>></td><td>ordered</td><td>bool</td></tr>
<tr><td>ordered</td><td>>=</td><td>ordered</td><td>bool</td></tr>
<tr><td>numeric or string</td><td>+</td><td>numeric or string</td><td>numeric or string</td></tr>
<tr><td>numeric</td><td>-</td><td>numeric</td><td>numeric</td></tr>
<tr><td>numeric</td><td>*</td><td>numeric</td><td>numeric</td></tr>
<tr><td>numeric</td><td>/</td><td>numeric</td><td>numeric</td></tr>
<tr><td>numeric</td><td>%</td><td>numeric</td><td>numeric</td></tr>
<tr><td>integer</td><td>|</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td>&</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td><<</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td>>></td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td>&^</td><td>integer</td><td>integer</td></tr>
<tr><td>integer</td><td>^</td><td>integer</td><td>integer</td></tr>
</table>

- http://golang.org/ref/spec#Arithmetic_operators
- http://golang.org/ref/spec#Comparison_operators
- http://golang.org/ref/spec#Logical_operators

3.5 Function call

```
expr(arg1, arg2, ..., argn)
```

A function call is well-typed if:

- expr is well-typed and has function type (T1 * T2 * ... * Tn) -> Tr;
- arg1, arg2, ..., argn are well-typed and have types T1, T2, ..., Tn respectively.

The type of a function call is Tr.

3.6 Indexing

```
expr[index]
```

Indexing into a slice or an array is well-typed if:

- expr is well-typed and has type Slice<T> or Array<int, T>;
- index is well-typed and has type int.

The result of the indexing expression is T.

Note: The Go specification states that the compiler should to report an error if the index of an array (not of a slice) evaluates to a statically-known constant that is outsides the bounds of the array. You do not have to implement this at compile-time in GoLite, instead we'll do the check at runtime.

3.7 Field selection

```
expr.id
```

Selecting a field in a struct is well-typed if:

- expr is well-typed and has type S;
- S is a struct type that has a field named id.

The type of a field selection expression is the type associated with id in the struct definition.

3.8 append

```
append(id, expr)
```

An append expression is well-typed if:

- id is found in the symbol table and maps to a Slice<T>;
- expr is well-typed and has type T.

The type of append is Slice<T>.

3.9 Type cast

```
type(expr)
```

A type cast expression is well-typed if:

- type is a int, float64, bool, rune, or a type alias that maps to one of those four;
- expr is well-typed and has a type listed in the previous bullet point.

The type of a type cast expression is type.