Milestone 1: Go Parser

Xavier Denis

March 6, 2015

1 Design Decisions

While Haskell provides a natural interface for many recursive tasks, building a typed AST and the requirements to return the symbol table posed a few challenges.

In the Haskell community the AST Typing problem has no definitive solution. Haskell itself uses an explicitly typed AST which gets built during typechecking. Other approaches include building an infinite type using Fixed points of the types and exploiting the Cofree-Comonad (whatever that means). The downside of these approaches is that it makes pattern matching much more difficult. Instead, I opted to use an intermediate approach, that is I parameterized all types containing 'Expression' over 'a'. By adding 'a' to all the constructors of 'Expression' I can attach any information I wish to expressions.

I chose to model the actual typechecker through another typeclass called 'Checkable'. This simplifies the code slightly, since all checker methods share a name. It fits nice into the semantics of typeclasses which are used to express a common behaviour shared by types.

The definition given above, which is used in the code causes a problem. The type 'a' has kind $* \rightarrow *$, however, Haskell does not support polymorphic

```
type Check = ExceptT TypeError (State (String, Counters, Env))
type Ann = Type
class Checkable a where
  check :: a -> (Check (a Ann))
```

instance declarations for higher kinded types by default. In my previous typeclasses (Pretty, Weeder), I exploited these polymorphic instances to simplify dealing with 'Maybe a' and '[a]'. This meant that the code would be slightly more verbose than was strictly required.

Returning the symbol table on scope exit causes a few problems of it's own, most notably it unreasonable to not use the 'State' monad to carry the log through the check. However, since the 'Either' monad was already being used to represent abortable computation, monad transformers were required to combine both monads together. While monad transformers are great tools, they can come with a performance penalty of up to 300%.

2 Scoping Rules

There are 3 major scoping cases:

1. Functions: When functions are created, we open a scope to push all the parameters into. Since function bodies are represented by a block statement and would normally open another scope, we manually read out the statements in the function to avoid parameter shadowing.
2. If/Switch/For: These statements can have statements in their condition clauses, so we open a scope to process these statements.
3. Block: Block statements are used to represent all blocks in curly braces, including If statements or For loops. These open a scope when they process and pop it when the close.

3 Type Rules

There are a *many* type rules used. Briefly they can fall into 3 major sections:

1. Type: These checks focused on properly adding types to the symbol table. Additionally, they deal with all the scoping issues that happen when nested types share the same name. Relevant tests:
   (a) INVALID/alias.go
   (b) INVALID/tricky_alias.go

(c) INVALID/invalid_if.go

(d) VALID/tricky_alias.go

(e) VALID/alias.go

(f) VALID/var_scopes.go

2. Expression: These checks focus on checking all expressions according the go and golite specs. They also make sure to assign the correct type to the expression so that future operations succeed. Relevant tests:

   (a) INVALID/invalid_comp.go

   (b) INVALID/assign.go

   (c) VALID/valid_slice.go *append is treated as a function invocation (expression)*

   (d) VALID/var_scopes.go

3. Statement: These checks are mainly focused on correct recursion. Certain statements such as for/if/switch also require secondary checks. Relevant tests:

   (a) INVALID/invalid_dec.go

   (b) INVALID/invalid_if.go

   (c) VALID/for.go

   (d) VALID/valid_slice.go

   (e) VALID/valid_switch.go

4 Work Split

I, Xavier Denis, did this entire milestone.