Symmetry layer specification

Mike Stay (stay@pyrofex.net)
2022-07-20

Overview

Types can be seen as assertions about the "shape" of data. Function types capture the shape of the input and output of a function. This specification describes "layers", another kind of assertion about functions. Layers assert limitations on the behavior of the function.

Suppose Alice has some proprietary data she would like to process with a library written by Bob. Bob has obfuscated the library, since he would like to keep its algorithms secret. How can Alice guarantee that the library won't leak her proprietary information to Bob? Alice either has to attempt to reverse engineer the library and prove to herself that it doesn't use the networking API—which, in the presence of dynamic linking, is undecidable—or Alice has to resort to running the software in a different process and sandboxing it.

Layers can help. By declaring a function to be in the Core layer and insisting that the methods on Alice's objects are also in the Core layer, Bob can guarantee to Alice that no matter what the implementation details are, that function cannot cause side effects: no variables get mutated, no nondeterministic operations like reading the time of day or iterating over a map's keys occur, no goroutines get forked, and no I/O occurs, including sending information over the network. The library simply won't compile if Bob attempts to do any of those things.

This is a somewhat contrived example, of course. More realistically, Alice would like to write code and prove that it behaves properly. She would like it to be easy to audit and easy to maintain. Layers are declarations about what code can't do, making it easier to reason about what it can.

The idea of layers comes from the Noether language design and from auditors, but the principal insight into how to implement the ideas in Go comes from Choudhury and Krishnaswami's paper. Whereas a purely functional language like Haskell can only cause side effects through the use of a monad, an imperative language like Go can restrict side effects through a comonad.

Each layer consists of a syntactic part and a predicate on (AST, symbol table) pairs. Alice and Bob use the syntactic part to make assertions in their source code. The symmetry tool uses the predicate as part of its type checker. Once it has verified the assertions, the layers are erased and have no runtime cost.

Capabilities

While a predicate can be any function of an AST subtree and a symbol table, the layers under consideration here are about deviations in behavior from pure functions: assignment, I/O, concurrency, nondeterminism, etc. Sometimes the authority to cause deviations is partially grammatical; for example, the assignment operator, increment and decrement, iteration over maps, go keyword, and the send and receive operators all require both a reference and a grammatical construct to cause a side effect. These are instances of rights amplification. Other times, authority to deviate from pure functional behavior is encapsulated in an impure function like the delete function on maps or the Println function. We can prevent the former by allowing a static sublanguage, filtering out ASTs that include the offending grammatical constructions. We can prevent the latter using layers as safety attestations, as described below.

Layers

The syntactic part of a layer has a corresponding generic struct with one type parameter:

```
type <LayerName>[X any] struct {
   data X
   <layerName> struct{}
}
```

a function for extracting the data:

```
func (d <LayerName>[X]) Data() X { return d.data }
```

and a function for constructing the struct:

```
func New<LayerName>[X any](data X) <LayerName>[X] {
        return <LayerName>{data, struct{}{}}
    }
```

Note that the field `<layerName>` begins with a lowercase letter, so these structs can only be created using the `New<LayerName>()` function. It also serves to prevent type conversion between these structs.

A programmer makes an assertion about an expression `data` by invoking the `New<LayerName>()` function on it. The resulting wrapper type is a guarantee that `data` satisfies the restrictions. Other functions can then restrict their arguments to values of that type; this serves as a kind of precondition on the input.

Each layer also has predicate with the following signature:

```
func <LayerName>Pred(subtree *ast.Tree, scope *analyze.SymbolTable) bool
```

where the lexical scope object contains the bindings visible to the subtree. The predicate runs at compile time on the AST of the argument to `New<LayerName>()` at each of its call sites.

Marking function and method declarations

We are most interested in values of type `<LayerName>[func(X)Y]`. Function literals appearing in `New<LayerName>()` calls can be checked *in situ*, but Go has no general `let` statement that would allow us to define a package-scoped value of type `<LayerName>[func(X)Y]` in the way we're able to with non-wrapped functions. Therefore we introduce new syntax to allow **marking** functions and methods with a layer annotation `_<layerName>`; note the lowercase initial of the annotation as compared to the uppercase generic type. We could derive the minimum layer that the code satisfies, but explicitly marking it reduces the potential for errors. An unmarked function is taken to be in the `_full` layer.

The mark comes immediately after the func keyword in a declaration, *e.g.*

```
func _core Foo(X) Y {...}
func _core (b Bar) Foo(X) Y {...}
```

The mark comes immediately before the name of a method in an interface, *e.g.*

```
type Foo interface {
        _core Bar(X) Y
}
```

Syntax examples

1. `func _<layerName> Foo(...) {...}`
   `func _<layerName> (r Receiver) Foo(...) {...}`
   These annotations mean the body of Foo should pass the associated predicate. The analyzer will produce an error if it does not.
2. `func _core Foo(f func(X)Y) {...}`
   This is a special case of 1. Because the body of `Foo` has no capabilities, it cannot by itself cause any side effects. However, the body can invoke `f`, which may have capabilities, and cause side effects that way.
3. `func _core Foo(f Core[func(X)Y]) {...}`
   This is a special case of 1. Because the body of `Foo` has no capabilities and `f` has no capabilities, invoking `Foo` cannot cause side-effects. It is not necessarily a pure function, however: it may close over a reference that some other closure could mutate. For example, `Foo` could be a getter with a corresponding `_stateful` setter.
4. `<LayerName>[X]`
   This is a Go generic type that serves to mark values of type `X` as having bounded authority. In order to create a value of this type, code has to say `New<LayerName>[X](x)`. The analyzer examines these sites and verifies that `x` satisfies the layer criteria (see the next section).
5. `<LayerName>[struct {...}]`
   This is a special case of 4. To create a value of this type, code has to say `New<LayerName>(x)`, where `x` is a struct literal of the given shape whose field types all satisfy the layer criteria.
6. `<LayerName>[interface {...}]`
   This is a special case of 4. To create a value of this type, code has to say `New<LayerName>(x)`, where `x` is a struct literal whose field types all satisfy the layer criteria and the given interface.
7. `type Foo interface { _<layerName> Bar(X) Y }`
   A struct implementing this interface has to have its receiver method `Bar` marked as `_<layerName>`.

Values that can be passed to New<LayerName>()

In order to say what values can be passed to `New<LayerName>()`, we first have to say what types they can be.

Types that satisfy `<LayerName>` are defined inductively as follows:
- bool, numeric, character, or string
- *X for X satisfying `<LayerName>`
- chan X for any X
- slice or array []X for X satisfying `<LayerName>`
- map[K]V for K, V satisfying `<LayerName>`
- a struct whose field types satisfy `<LayerName>`

- `<LayerName'>[X]` for any X and any layer `<LayerName'>` implying `<LayerName>`

Note that interfaces do not satisfy layers, but when X is an interface type, `<LayerName>[X]` satisfies `<LayerName>` under the last criterion above. Similarly, function types don't satisfy layers, but when X is a function type, `<LayerName>[X]` satisfies `<LayerName>`.

Values that satisfy `<LayerName>` are defined as follows:

- a value of a type satisfying `<LayerName>`
- an expression whose free variables are each of a type satisfying `<LayerName>`, and whose code passes the predicate. Note that free variables in a function expression include method names and package-qualified functions, but not parameters to the function
- a declared function or method such that the free variables of the body are each of a type satisfying `<LayerName>`, and the code of the body passes the predicate

Core

Core forbids invoking functions or methods unless they have been checked as satisfying Core. That is, it can only invoke

- functions declared to be _core:

```
func _core Foo(x X) Y {...}
```

- methods declared to be _core:

```
func _core (f Foo) Bar(x X) Y {...}
```

- Data property of values of type Core[func(X)Y]:

```
func _core Foo(x X) Y { ... }

func _core Bar(x X) Y {
   // NewCore() call site gets checked
   // Foo is marked _core, so it's OK
   foo = NewCore[func(X)Y](Foo)
   // Data property of a Core[] object
   // is safe to invoke
   return foo.Data()(x)
}
```

It also forbids mutation, goroutines, nondeterminism, and invoking non-core functions other than parameters. More specifically, it forbids

- assignment except when assigning the initial value as part of a variable declaration
- increment, decrement

- the go keyword
- send and receive operators
- iterating over maps (though we may provide a map library that tracks insertion order)
- user input
- time of day
- any other source of nondeterminism we discover

Note that although `chan X` satisfies Core, Core code (and Stateful code, see below) cannot make use of a channel other than to pass it around. Similarly, Core code can read from but not write to arrays, slices, structs, and maps.

Stateful

Stateful has the restrictions of Core, but allows assignment and increment/decrement. Nondeterminism is still forbidden. Core implies Stateful.

Full

Full has the restrictions of Stateful, but allows using send and receive operators and iterating over maps. Nondeterminism on the blockchain is still undecided. To implement this, we would need to do one of the things described in the language extension doc. Core and Stateful imply Full.

Optimization

If everything typechecks properly with the calls to `New<LayerName>` and uses of `Data` fields in place, then the code satisfies the constraints and we can remove all references to them. We then typecheck the newly emitted code; this is to prevent a malicious user from writing something like the following:

```
// Alias the function whose callsites are checked
FakeNewCore := NewCore[func(a ...any) (n int, err error)]
// Invoke the alias to create a counterfeit object
Bad = FakeNewCore(fmt.Println)
```

Because the analyzer only looks at call sites for `NewCore`, it doesn't look at the call site for `FakeNewCore`. The value `Bad` is of type `Core[func(a ...any) (n int, err error)]`, but `fmt.Println` is never checked, breaking the guarantee that `Core[]` is supposed to provide. However, by removing all references to `NewCore`, the modified definition of `FakeNewCore` results in a syntax error.

Examples

See the eng/symmetry-layerspec repo for examples.