1 Data Structures

For each interface object, we maintain a set of its defined methods: F.method_set.

Maintain the following two hash tables or dictionaries:

1. hash_table1: {Fi : {T1 : {M1, … Mk}, T2 : {...}, … Tj : {...}}} where the key is an interface Fi, the value is another hash table, whose key is a type Tj, whose value is a set of methods Mk's implemented by Tj;
2. hash_table2: {Mm : {F1, F2, … Fn}} where the key is a method Mm, the value is its corresponding set of interfaces Fn's

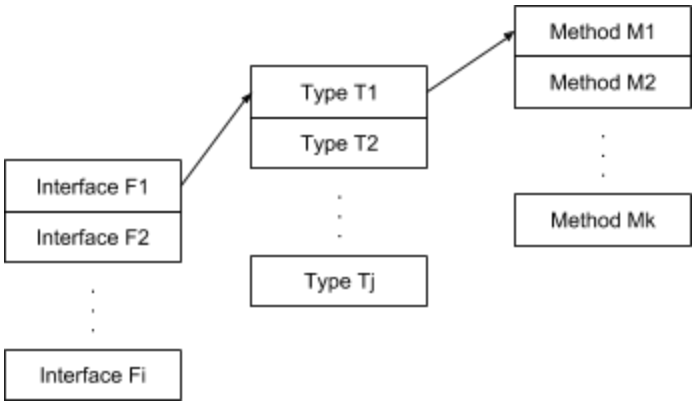

Figure 1: hash_table1

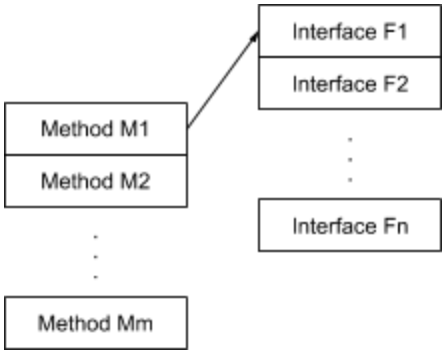

Figure 2: hash_table2

2 Parser Functions

1. When encountering a new interface F, update F.method_set to include all its defined methods M1, … Mk. Then add {M1, F}, … {Mk, F} key-value pairs to the sets in hash_table2:

   hash_table2[Mk].add(F)

   The time complexity of this step is bounded to O(n*m) where n is the # of interfaces and m is the most # of methods that an interface can have.

2. When encountering a method M implementation for Type T. First, look up hash_table2 to find the set of interfaces the method belongs to. Then for each F in {hash_table2[M]}, update hash_table1:

   hash_table1[F][T].add(M)

   The time complexity of this step is the same as the previous step, which is bounded to O(n*m).

3. When using a type T as interface F. We need to ensure that T implements all the methods required by F. In this case we check the size of hash_table1[F][T]:

   if (len(hash_table1[F][T]) < len(F.method_set)):

   report "not implemented all required methods" error

   The time complexity of this step is O(j), where j is the total # of the cases using T as F.

4. When calling a method M of type T as interface F. We check if M is in F.method_set. If the case holds we can call method M successfully.

   if (M not in F.method_set):

   report “function M not defined in F” error

   else:

   call hast_table1[F][T][M]

   The time complexity of this step is O(r), where r is the total # of method calls.

3 An Example

Let us go through a test case found on: https://tour.golang.org/methods/9

```
package main
import (
	"fmt"
	"math"
)

type Abser interface {
	Abs() float64
}

func main() {
	var a Abser
	f := MyFloat(-math.Sqrt2)
	v := Vertex{3, 4}

	a = f  // a MyFloat implements Abser
	a = &v // a *Vertex implements Abser

	// In the following line, v is a Vertex (not *Vertex)
	// and does NOT implement Abser.
	//a = v

	fmt.Println(a.Abs())
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
	if f < 0 {
		return float64(-f)
	}
	return float64(f)
}

type Vertex struct {
	X, Y float64
}

func (v *Vertex) Abs() float64 {
	return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

1. When reaching line 8 “`type Abser interface`”:
   Abser.method_set = {Abs}
   for M in Abser.method_set:
   hast_table2[M].add(Abser)

2. When reaching line 29 “`func (f MyFloat) Abs() f loat64`” and line 40 “`func (v *Vertex) Abs() float64`”:
   for F in hash_table2[Abs]:
   hash_table1[F][MyFloat].add(Abs)
   for F in hash_table2[Abs]:
   hash_table1[F][*Vertex].add(Abs)

3. When seeing line 17 and 18 “`a = f`” and “`a = &v`”:
   if (len(hash_table1[Abser][MyFloat]) < len(Abser.method_set)):
   report “not implemented all required methods” error
   if (len(hash_table1[Abser][*Vertex]) < len(Abser.method_set)):
   report “not implemented all required methods” error
   In both cases, the checks go through without reporting an error.

4. When calling “`a.Abs()`” at line 24:
   if (Abs not in Abser.method_set):
   report “function Abs not defined in Abser” error
   else:
   call hash_table1[Abser][*Vertex][Abs]
   In this case, we make a successful function call.