Iteration in Go

Abram Hindle
abram.hindle@ualberta.ca
Department of Computing Science
University of Alberta
http://softwareprocess.es/

Warning

- This talk is not idiomatic
- Also it is written by a GO Impostor ;-)
- Wish I saw Nathan's talk before I did some of these examples :(

Iteration

- Programming is useful because we can repeat tasks
- Iteration is one of the fundamental building blocks of most programming languages
- Iteration usually refers to repetition
  - In mathematics iterating a functions means applying it repeatedly
  - In programming iteration can refer to any method of repetition.

Concepts

- Block
  - A chunk of code
  - Usually the part that gets repeated
  - Usually follows scope rules
- Condition
  - An expression that evaluates to true or false
  - Commonly used to determine if a loop continues

Concepts

- In Order – Execute in Sequence
- Out of Order – Execute in any order
- Dependency – One value depends on another

~~While~~ For Loops

- Repeats a block until a condition is met.
- Loop invariant executes first

```
condition := false
for condition {
    // ...
    condition = !condition
}
reader := bufio.NewReader(os.Stdin)
reads := 0
for {
    reads++
    _, err := reader.ReadString('\n');
    if err!=nil {
        break;
    }
}
fmt.Printf("Reads: %d\n", reads)
```

```
x := 10
for x > 0 {
    x = x - 1
}
// x is 0
fmt.Printf("x is %d\n", x)

// maybe you're not sure how
// many iterations you need?
y := 100.0
for y > 1 {
    y = y / 3;
}
fmt.Printf("y is %f\n", y)
// y is 0.41152263374485604
```

For Loop

- Iterate with a condition or over a collection of elements.

```
sum := 0;
for i := 0 ; i < 10; i++ {
    sum += i;
}
fmt.Printf("Sum is %d\n", sum)

s := ""
s2 := "
v := []string{"a","b","c"}
u := map[string]string{"A":"a", "B":"b","C":"c"}
for i, val := range v {
    for j := 0; j <= i; j++ {
        s += val;
    }
}
fmt.Printf("s is [%s]\n",s)
```

```
for key, val := range u {
    s += key;
    s += val;
}
fmt.Printf("s is [%s]\n",s)
for i, val := range s2 {
    // i is the byte location
    for j := 0; j <= i; j++ {
        s += string(val);
    }
}
fmt.Printf("s is (note that the
 unique was base-4) [%s]\n",s)
```

Recursion

- Arbitrary flow control
- Good for iterating datastructures like trees
- Watch out for stackoverflows!

```
stump := map[string]Tree{}

tree := Tree{0,
    map[string]Tree{
        "a":Tree{1,stump},
        "b":Tree{2, map[string]Tree{
                "h":Tree{8,stump},
                "i":Tree{9,stump},
                "j":Tree{10,stump},
            },
        },
        "c":Tree{3,stump},
        "d":Tree{4, map[string]Tree{
                "e":Tree{5,stump},
                "f":Tree{6,stump},
                "g":Tree{7,stump},
            },
        },
    },
}
```

Recursion

- Arbitrary flow control
- Good for iterating datastructures like trees
- Watch out for stackoverflows!

```
fmt.Printf("Treesum %d\n", TreeSum( tree ))

for i := range iInt(6).Iter() {
    fmt.Printf("Wow! %v\n", i)
}

// Call Back walker
TreeWalker(tree, func(t Tree) {
   fmt.Printf("Node value %d\n",t.Value) } )
```

Recursion

```
type Tree struct {
     Value int
     Branches map[string]Tree
}


func TreeSum( tree Tree ) int  {
    sum := tree.Value;
    for _, val := range tree.Branches {
        sum += TreeSum(val)
    }
    return sum;
}
```

```
func TreeWalker( tree Tree,
                       f func(Tree) )  {
    f(tree)
    for _, val := range
                       tree.Branches {
        TreeWalker(val,f)
    }
}
```

Channel Iterators

```
type iInt int

func (max iInt) Iter () <-chan iInt {
    ch := make(chan iInt);
    go func () {
        m := int(max)
        for i := 0; i <= m; i++ {
            ch <- iInt(i)
        }
        close(ch)
    } ();
    return ch
}
```

Channel Iterators

```
func TreeIter( tree Tree ) <-chan Tree {
    ch := make(chan Tree);
    go func () {
        TreeWalker( tree, func(t Tree) {
            ch <- t
        })
        close(ch)
    } ();
    return ch
}

func (tree Tree) Iter() <-chan Tree {
    return TreeIter( tree )
}
```

Channel Iterators

```
for i := range iInt(6).Iter() {
    fmt.Printf("Wow! %v\n", i)
}

// Call Back walker
TreeWalker(tree, func(t Tree) { fmt.Printf("Node value
%d\n",t.Value) } )

for tree := range tree.Iter() {
    fmt.Printf("Now via Iter Node value %d\n",tree.Value)
}
```

OO Iteration (Iterators)

- Object with a `Next()` method and Value()

```go
type StringIterator struct {
    current int
    s []rune
}
```

```go
func Iterator(s string) *StringIterator {
    return &StringIterator{
        current: -1, s: []rune(s)}
}
```

```go
// http://ewencp.org/blog/golang-iterators/
func (si *StringIterator) Next() bool {
    si.current++
    return (si.current < len(si.s))
}
func (si *StringIterator) Value() string {
    return string(si.s[si.current])
}
```

OO Iteration (Iterators)

- Object with a `Next()` method and Value**()**

```
type EvenStringIterator struct {
    current int
    s []rune
}
func (si *EvenStringIterator) Next() bool {
    si.current += 2;
    return (si.current < len(si.s))
}

func (si *EvenStringIterator) Value() string {
    return string(si.s[si.current])
}


func EvenIterator(s string) *EvenStringIterator {
    return &EvenStringIterator{current: -1, s: []rune(s)}
}
```

OO Iteration (Iterators)

- Object with a `Next()` method and Value()

```
si := Iterator(s)
for si.Next() {
    fmt.Printf("String val! %s\n", si.Value())
}

si2 := EvenIterator(s)
for si2.Next() {
    fmt.Printf("Even String val! %s\n", si2.Value())
}
```

Order

- Did you notice something?
- Everything iterated in order.
- But what if order doesn't really matter?

Map

- In mathematics iterating a functions means applying it repeatedly
- A map function applies 1 function to all elements in a collection and produces a new collection of the results of that function
  - Usually this is in order
  - But you don't have to do it in order
- Map f (a,b,c) => (f(a), f(b), f(c))
- Map square (1,2,3) => (1,4,9)

Map Example

```go
func intIntMap( iarr []int, cb (func(int) int)) []int {
    out := make( []int, len(iarr))
    for i,v := range iarr {
        out[i] = cb( v )
    }
    return out
}

// MACROS??? GENERICS???
func strStrMap( iarr []string, cb (func(string) string)) []string {
    out := make( []string, len(iarr))
    for i,v := range iarr {
        out[i] = cb( v )
    }
    return out
}
```

Map Example

```
v2 := []int{1,2,3,4,5,6,7,8}
inc := func(x int) int { return 1 + x }
sqr := func(x int) int { return x * x }
// lack of generics
v3 := intIntMap(v2, inc)
fmt.Printf("inc v2: [%v] v3: [%v]\n",v2,v3)
// lack of generics
v3 = intIntMap(v2, sqr)
fmt.Printf("sqr v2: [%v] v3: [%v]\n",v2,v3)

basename := func(path string) string {
    sp := strings.Split(path,"/")
    return(sp[len(sp) - 1])
}
vs := []string{"/home","/file", "/usr/local"}
vs2 := strStrMap( vs, basename )
fmt.Printf("basename vs: [%v] vs2: [%v]\n",vs,vs2)
```

Map Example

```
urls := []string{"http://cbc.ca", "http://gc.ca", "http://alberta.ca"}
status := func( uri string ) string {
    resp, _ := http.Get(uri)
    return(resp.Status)
}

statuses := strStrMap(urls, status);
fmt.Printf("statuses: %v\n", statuses)
```

Parallelism with Map

- Think in “map” -- Think Parallel
- Limit dependencies of a block in order to parallelize the computation!

```
mySum := func(l []int) int {
    sum := 0
    for _,v := range l {
        sum += v
    }
    return(sum)
}
sumres := mySum( intIntMap(series(1,1000),sqr))
fmt.Printf("sumres: %v\n", sumres)
psumres := mySum(parallelIntIntMap( series(1,1000), sqr, 4))
fmt.Printf("psumres: %v\n", psumres)
```

Parallelism with Map (Continued)

- IO is slow and inherently parallelizable!

```
pgets := parallelStrStrMap( urls, status, 3)
fmt.Printf("Stupid examples with URLs %v\n", pgets)
```

Parallelism with Map (Continued)

```
func parallelIntIntMap( l []int, f (func(int) int),
workers int) []int {
    chans := make( [](chan []int),  workers )
    for i := range chans {
        chans[i] = make(chan []int)
    }
    unit := len(l)/workers
    for i := 0 ; i< workers; i++ {
        mychan := chans[i]
        start := i * unit
        end := (i + 1)*unit
        if end >= len(l) {
            end = len(l)
        }
        subl := l[start:end]
```

Parallelism with Map (Continued)

```
for i := 0 ; i< workers; i++ {
    mychan := chans[i]
    start := i * unit
    end := (i + 1)*unit
    if end >= len(l) {
        end = len(l)
    }
    subl := l[start:end]
    par := func(l []int) {
        mychan <- intIntMap(l, f)
        close(mychan)
    }
    go par(subl)
}
out := make([]int, len(l))
for i := 0; i < workers; i++ {
    arr := <- chans[i]
    start := i*unit
```

Parallelism with Map (Continued)

```
    out := make([]int, len(l))
    for i := 0; i < workers; i++ {
        arr := <- chans[i]
        start := i*unit
        end := start + len(arr)
        copy(out[start:end], arr)
    }
    return out
}
```

Reduce

- Linear, 1 at a time
- Collapse a collection in a single value via an operator or function of 2 args
  - f(e1, f(e2, f(e4, ... f(e99,e100)))...)))
  - add(e1, add(e2, add(e4, ... add(e99,e100)))...)))
- Sum is a reduce

```
vex := []int{1,2,3,4,5,6,7,8,9};
iadd := func(x, y int) int { return(x + y) }
vexr := intIntReduce(vex, iadd)
fmt.Printf("Sum: %v\n", vexr)
```

Reduce

```
func intIntReduce( iarr []int, cb (func(int,int) int))
     int {
   o := iarr[0]
   m := len(iarr)
   for i := 1; i < m; i++ {
      o = cb( iarr[i], o)
   }
   return o
}
```

Reduce Can Be Parallel (Sometimes)

```
preduced := parallelIntIntMapReduce( series(1,100000000),
                                     sqr, iadd, 4)
fmt.Printf("Reduced %v\n", preduced)
```

Reduce Can Be Parallel (Sometimes)

```
func parallelIntIntMapReduce( l []int,
        mapper (func(int) int),
        reducer (func(int,int) int),workers int) int {
    chans := make( [](chan int),  workers )
    //var chans [workers]chan []int
    for i := range chans {
        chans[i] = make(chan int)
    }

    unit := len(l)/workers
```

Reduce Can Be Parallel (Sometimes)

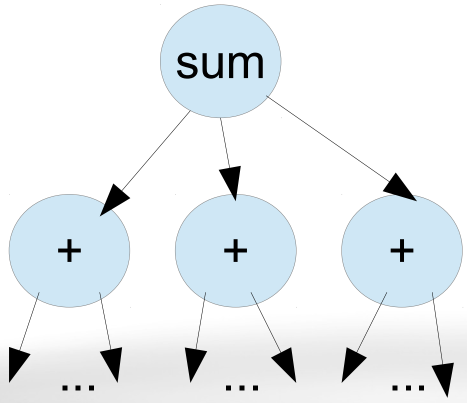
```
for i := 0 ; i< workers; i++ {
   mychan := chans[i]
   start := i * unit
   end := (i + 1)*unit
   if end >= len(l) {
      end = len(l)
   }
   subl := l[start:end]
   par := func(l []int) {
      mychan <- intIntReduce(intIntMap(l, mapper), reduc
      close(mychan)
   }
   go par(subl)
}
```

Reduce Can Be Parallel (Sometimes)

```
    reductions := make( []int, workers)
    for i := 0; i < workers; i++ {
        r := <- chans[i]
        reductions[i] = r
    }
    return intIntReduce(reductions, reducer)
}
```

Trees, Communtativeness and Initialization

- Can your problem be modelled as a TREE?
- Problems with commutative or associative parts can often be modelled as a tree of computation.
- Different branches may be executed in Parallel.
- One can reduce dependencies by avoid initialization (e.g. sum = 0)



Generic Map and Reduce and Go

- We can use an interface to give some collections Map like ability, but that's irritating

```
type ints []int


func (iarr ints) Map(cb (func(E)E)) ([]E) {
   out := make( []E, len(iarr))
   for i,v := range iarr {
      out[i] = cb( v )
   }
   return out
}
```

Generic Map and Reduce and Go

- We can use an interface to give some collections Map like ability, but that's irritating

```
evex := ints(vex).Map( func(x E) E {
   return (x.(int)+1)
})
fmt.Printf("Generic Map: %v\n", evex)
evex2 := ints(vex).Map( func(x E) E {
   return (float32(x.(int))+1.1)
})
fmt.Printf("Generic Map: %v\n", evex2)
```

```
Generic Map: [2 3 4 5 6 7 8 9 10]
Generic Map: [2.1 3.1 4.1 5.1 6.1 7.1 8.1 9.1 10.1]
```

Generic Map and Reduce and Go

- We can abuse the empty interface{}

```
type Mapper interface {
   Map( (func(interface{})interface{}) ) ([](interface{}))
}

type Reducer interface {
   Reduce( (func(interface{},interface{})interface{}) )
                                              (interface{})
}

type E interface{}
```

Generic Map and Reduce and Go

- We can abuse the empty interface{} -> E
- But can we actually call this?

```
type Collection []E
func (iarr Collection) Map(cb (func(E)E)) ([]E) {
   out := make( []E, len(iarr))
   for i,v := range iarr {
      out[i] = cb( v )
   }
   return out
}
```

Generic Map and Reduce and Go

- We can abuse the empty interface{} -> E
- But can we actually call this?

```
type Collection []E
func (iarr Collection) Map(cb (func(E)E)) ([]E) {
   out := make( []E, len(iarr))
   for i,v := range iarr {
      out[i] = cb( v )
   }
   return out
}
```

Generic Map and Reduce and Go

- But can we actually call this?
- Not really, we need to convert to a collection :(

```go
svex := []string{"a","b","c"}
sout := collection(svex).Map( func(x E) E {
   return (x.(string) + x.(string))
})
fmt.Printf("Generic Map w/ String: %v\n", sout)
sout2 := collection(svex).Map( func(x E) E {
   return len(x.(string))
})
fmt.Printf("Generic Map w/ String: %v\n", sout2)
evex3 := collection(vex).Map( func(x E) E {
   return (x.(int)+1)
})
fmt.Printf("Generic Map: %v\n", evex3)
```

Generic Map and Reduce and Go

- But can we actually call this?
- Not really, we need to convert to a collection :(

```
svex := []string{"a","b","c"}
```

```
Generic Map: [2 3 4 5 6 7 8 9 10]
Generic Map: [2.1 3.1 4.1 5.1 6.1 7.1 8.1 9.1 10.1]
Generic Map w/ String: [aa bb cc]
Generic Map w/ String: [1 1 1]
Generic Map: [2 3 4 5 6 7 8 9 10]
```

```
evex3 := collection(vex).Map( func(x E) E {
    return (x.(int)+1)
})
fmt.Printf("Generic Map: %v\n", evex3)
```

Generic Map and Reduce and Go

- But can we actually call this?
- Not really, we need to convert to a collection :(

```
// https://stackoverflow.com/questions/12753805/type-converting-slices-of-interfaces-in-go/12754757#12754757
func collection(sliceOfStuff E) Collection {
   ourSlice := reflect.ValueOf(sliceOfStuff)
   out := make([]E, ourSlice.Len())
   for i := 0 ; i < ourSlice.Len(); i++ {
      out[i] = ourSlice.Index(i).Interface()
   }
   return out
}
// interface{} is a value and a type tag (new obj)
```

Generic Map and Reduce and Go

- Now we can write these nice helper functions

```
func Map(coll E, cb (func(E)E)) ([]E) {
   return collection(coll).Map(cb)
}
func (iarr Collection) Reduce(cb (func(E,E)E)) E {
   o := iarr[0]
   m := len(iarr)
   for i := 1; i < m; i++ {
      o = cb( iarr[i], o)
   }
   return o
}
func Reduce(coll E, cb (func(E,E)E)) E {
   return collection(coll).Reduce(cb)
}
```

Generic Map and Reduce and Go

- To evaluate it

```
slowMapRes := Map( vex, func(x E) E {
   y := x.(int)
   return (y*y)
})
fmt.Printf("Generic Map function: %v\n", slowMapRes)

slowMapReduce := Reduce( slowMapRes, func(x E, o E) E {
   return x.(int) + o.(int)
})
fmt.Printf("Generic Reduce function: %v\n", slowMapReduce)
```

```
Generic Map function: [1 4 9 16 25 36 49 64 81]
Generic Reduce function: 285
```

Conclusions

- Main forms of Go iteration:
  - For / Iterators / Range / Recursion / Map / Reduce
  - Generics are possible but come with a cost
- Reducing dependencies in blocks allows iteration to be parallelized.
- Consider if order or strictness can be are actually needed?
- These concepts apply to other languages as well.

Resources

- Go Spec http://golang.org/ref/spec
- Eleanor McHugh - Going Loopy: Iteration in Go
  https://www.youtube.com/watch?v=RFIOSjkB-j8
  http://www.slideshare.net/feyeleanor/presentation-289
- https://stackoverflow.com/questions/12363030/read-from-initial-stdin-in-
- Go Language Patterns https://sites.google.com/site/gopatterns/object-oriented/iterators
- Iterators in Go http://ewencp.org/blog/golang-iterators/
- Map in Go https://groups.google.com/forum/#!topic/golang-nuts/RKymTuSCHS0

Labels and breaks

- Label1:
- break Label1 // break the the loop at label1
- Label2:
- continue Label2 //advance next loop with Label2 label

Goto

- goto Label2 // goto Label2 if it isn't in a block

  // must be same scope

Go Channel Iterators

- See https://sites.google.com/site/gopatterns/object-or
-