Implementing a Lazy Generator

This lesson gives detailed knowledge on generators and lazy evaluation. It explains how to write a generator for lazy evaluation.

WE'LL COVER THE FOLLOWING

Introduction

A **generator** is a function that returns the next value in a sequence each time the function is called, like:

```
generateInteger() => 0
generateInteger() => 1
generateInteger() => 2
....
```

It is a producer that only returns the next value, not the entire sequence. This is called _lazy evaluations, which means only computing what you need at the moment, therefore saving valuable resources (memory and CPU). It is technology for the evaluation of expressions on demand.

Explanation

An example (of lazy generation) would be the generation of an endless sequence of even numbers. To generate it and use those numbers one by one would be difficult and certainly would not fit into memory! But, a simple function per type with a channel and a goroutine can do the job.

For example, in the following program, we see a Go implementation with a channel of a generator of ints. The channel is named `yield` and `resume`, terms commonly used in coroutine code.

```
package main
import (
"fmt"
)

var resume chan int
func integers() chan int {
  yield := make (chan int)
  count := 0
  go func () {
    for {
      yield <- count
      count++
    }
  } ()
  return yield
}

func generateInteger() int {
  return <-resume
}

func main() {
  resume = integers()
  fmt.Println(generateInteger()) //=> 0
  fmt.Println(generateInteger()) //=> 1
  fmt.Println(generateInteger()) //=> 2
}
```

Lazy Evaluation

In the code above, at **line 6**, `resume` is declared as a channel of integers. This is initialized at **line 24**, where it gets the return value of the `integers()` function.

Look at the header of `integers()` function at **line** 7. At **line 8**, it makes a channel `yield`, which will be returned at **line 16** to `resume`. It populates the channel by starting a goroutine with an *anonymous* function. This contains an infinite for-loop (from **line 11** to **line 14**), which puts successive integers on the channel.

Then, from **line 25** to **line 27**, back in `main()`, the function `generateInteger()` is called successively. As we see, **line 20** takes one value from the channel `resume`, and returns it. This prints out the numbers **0**, **1** and **2**.

A subtle difference is that the value read from the channel could have been

A subtle difference is that the value read from the channel could have been generated a while ago, rather than at the time of reading. If you need such behavior, you have to implement a *request-response* mechanism. When the generator's task is computationally expensive, and the order of generating results does not matter, then the generator can be parallelized internally by using goroutines. However, be careful that the overhead generated by spawning many goroutines does not outweigh any performance gain.

These principles can be generalized, by making clever use of the empty interface, closures and higher-order functions. We can implement a generic builder `BuildLazyEvaluator` for the lazy evaluation function (this should best be placed inside a utility package). The builder takes a function that has to be evaluated and an initial state as arguments and returns a function without arguments, returning the desired value. The passed evaluation function has to calculate the next return value as well as the next state based on the state argument. Inside the builder, a channel and a goroutine with an endless loop are created. The return values are passed to the channel from which they are fetched by the returned function for later usage. Each time a value is fetched, the next one will be calculated.

The following code is an implementation of the mentioned concept:

```
package main
import (
"fmt"
)

type Any interface{}
type EvalFunc func(Any) (Any, Any)

func main() {
    evenFunc := func(state Any) (Any, Any) {
        os := state.(int)
        ns := os + 2
        return os, ns
    }

    even := BuildLazyIntEvaluator(evenFunc, 0)
    for i := 0; i < 10; i++ {
        fmt.Printf("%vth even: %v\n", i, even())
    }
}

func BuildLazyEvaluator(evalFunc EvalFunc, initState Any) func() Any {
    retValChan := make(chan Any)
    loopFunc := func() {
        var actState Any = initState
        var retVal Any
        for {
            retVal, actState = evalFunc(actState)
            retValChan <- retVal

        }
    }
    retFunc := func() Any {
        return <-retValChan
    }
    go loopFunc()
    return retFunc
}

func BuildLazyIntEvaluator(evalFunc EvalFunc, initState Any) func() int {
    ef := BuildLazyEvaluator(evalFunc, initState)
    return func() int {
        return ef().(int)
    }
}
```

General Lazy Evaluation

This example uses several higher-order function techniques and is highly abstract. You can safely skip it on first reading if you want to.

Line 6 defines an alias `Any` for the *empty interface*. **Line 7** defines a function type `EvalFunc` that takes an `Any` and returns a tuple of two `Any’s`. In `main()` (from **line 10** to **line 14**), an *anonymous* function with type `EvalFunc` is defined and assigned to the name `evenFunc`. **Line 11** effectively does a conversion to *int* assigning to integer `os`, which is returned together with `os + 2` at the end of the function.

`BuildLazyIntEvaluator(evenFunc, 0)` is called at **line 16**, and assigned to `even`. The `even()` is a function, so it is called **10** times in the loop (see **line 18**), and the result of its call is printed.

Now, look at the header of `BuildLazyIntEvaluator` at **line 39**. It takes a function of type `EvalFunc` as a parameter, as well as a variable of any type `initState`. Its return value is a function that has no parameters and returns an int. It calls on its turn `BuildLazyEvaluator` at **line 40**. The result is assigned to `ef`. Then, we return an *anonymous* function that returns an integer, namely ef converted to *int*, at **line 42**.

Now, look at the header of `BuildLazyEvaluator` at **line 22**. It is the generalized version of `BuildLazyIntEvaluator`, which works for any type. At **line 23**, a channel `retValChan` of type `Any` is made. Then, we define an anonymous function at **line 24**, assigned to `loopFunc`. At **line 28** (which is contained in an infinite for loop) we see this call: `retVal, actState = evalFunc(actState)`. The `evalFunc` is the `even` function, so the return values are an integer and **integer + 2**. The integer is put on the channel `retValChan`, and the 2nd integer `actState` becomes the new first integer.

`loopFunc` is a function, which is started in a goroutine at **line 35**. Just before that at **line 32**, we defined a function as an *anonymous* function that gets and returns a value from `retValChan`. `retFunc` is then returned from `BuildLazyEvaluator` at **line 36**. The combined result of all this code is that we generate a series of even numbers in a lazy way with goroutines.

That is it on lazy generators. In the next lesson, you'll learn how to handle an anonymous function with a channel.