Go Escape Analysis Flaws

Dmitry Vyukov, @dvyukov
Feb 10, 2015

Abstract

The default Go compiler (gc) does escape analysis to identify objects that can be allocated on stack. This optimization is very useful because stack allocations are significantly cheaper for both memory allocator and garbage collector. However, there are some cases where escape analysis does very coarse analysis or gives up entirely. This leads to suboptimal results. The goal of this document is to identify such cases.
There are two ways to address the deficiencies: we either can do incremental improvements to the current algorithm fixing them one-by-one; or do a wholesale replacement of the current algorithm with a new, better one which addresses the deficiencies on a general basis. This decision is yet to be made and is outside of the scope of this document.

Closure calls

From issue: https://github.com/golang/go/issues/7714

```
var y int  // BAD: y escapes
func(p *int, x int) {
    *p = x
}(&y, 42)

x := 0  // BAD: x escapes
defer func(p *int) {
    *p = 1
}(&x)
```

Both snippets suffer from the same issue: closure calls are not analyzed.
Proposed fix: https://go-review.googlesource.com/#/c/4115/

Dot-dot-dot arguments always escape

Consider the following code:

```
func noescape(y ...interface{}) {
}

func main() {
    x := 0 // BAD: x escapes
    noescape(&x)
}
```

Escape analysis could track flow of &x through y.
https://github.com/golang/go/issues/7710

Assignment to indirection escapes

Whenever something is assigned to an indirection, it escapes. For example:

```
i := 0
pp := new(*int)
*pp = &i // BAD: i escapes
_ = pp
```

This particularly hits us with pointer receivers. And also introduces an unpleasant difference in the following program:

```
i := 0
var v X
v.p = &i // i does not escape
y := new(X)
y.p = &i // BAD: i escapes
```

General solution of this issue requires complex may-point-to analysis to find all objects that a pointer can potentially point to.

However, there is a relatively simple solution for a subset of this issue. Namely, if a pointer is assigned only once, then we can easily infer what it points to. This would allow to track flow through pointer receivers and eliminate difference between declaring a struct variable or a pointer to struct variable (see above example).
Here is a prototype of limited flow through indirections:
https://go-review.googlesource.com/#/c/3886

Assignments to slices and maps

This is related to "Flow through indirections" issue. We can allocate a map on stack, but any keys/values inserted into such map still escape. For example, consider:

```
m := make(map[int]*T) // the map does not escape
m[0] = new(T) // BAD: new(T) escapes
```

Full solution of this issue would require the same may-point-to analysis. But the case above is much simpler: we see that m always points to the same non-escaping map, so we can link inserted keys/values to the map itself.
This should also cover slice and map literals (https://github.com/golang/go/issues/8972).

Flow through function arguments

Escape tags can only describe flow of input arguments to return values. But they do not describe flow of input arguments to input arguments. For example, consider:

```
func (x *X) SetName(s string) {
    x.name = s // BAD: s contents escape to sink
}
```

This function makes s contents escape in the caller, because we can't express that s flows to x.
The proposed solution is to extend tag to encode flow of input arguments to both input arguments and return values.
An interesting example of this is: https://github.com/golang/go/issues/7213

Indirection level of arguments

Argument tags encode limited information. They can encode that a parameter flows to a result or that parameter contents flow to unknown result. They can't encode that a parameter flows with indirection level N to a particular result, nor that a parameter flows with indirection level N to sink. For example:

```
func foo(p **int) {
        sink = *p
}

func main() {
    x := new(int)
    foo(&x) // BAD: x escapes
}

var sink interface{} // In all subsequent examples sink means the
same.
```

Here both new(int) and x escape. While x should not escape.
The proposed solution is to encode exact levels in tags. For example, param x flow to result y with level 0 and param x flow to sink with level 1.
Part of the fix is in https://go-review.googlesource.com/#/c/3753/

Indirection level is wrong for assignments

Consider the following code:

```
     var v X
     p := &Y{} // BAD: &Y{} escapes
     v.f = *p
     return &v
```

Currently &Y{} literal is allocated on heap because indirection level is computed incorrectly when following escape flow. The flow is:

~r1 <- &v
v <- *p
p <- &Y{}

~r1 has level 0.
Then after &v level is decremented to -1.
Then after *p level is incremented to 0.
So we reach &Y{} with level 0, which means
that address &Y{} can reach ~r1 (escapes).

The flaw here is that we should not follow flow with level<0. Level<0 means that address of SRC can reach DST. However, if we store FOO to SRC, it does not mean that now address of _FOO_ can reach DST. It is still FOO content that can reach DST.

The solution is to resets level to 0 when we are following flow.
The following change has the fix:

https://go-review.googlesource.com/#/c/3890/

Indirection level is wrong for operations on interfaces

Consider the following code:

```
    v := X{}
    var x Iface = v
    x.Method() // BAD: makes interface backing storage escape
    sink = x.(X) // BAD: makes interface backing storage escape
```

The problem here is that we ignore implicit indirections that are present here. What effectively happens is:

```
    v := X{}
    var x Iface
    x.Type = _type_X
    x.Value = new(X) // It is this new(X) that escapes
    *x.Value = v
    (*x.Value).Method()
    sink = (*x.Value)
```

Is we model this exact implementation, then new(X) would not escape.
Proposed fix: when we convert a variable to interface, we need to introduce flow x <- new(X) and x <- &v (through a fake OADDR node). When we extract the value from the interface in sink = x.(X) (or call a method on interface) we need to introduce flow sink = *x (through a fake OIND node).
A proposed fix: https://go-review.googlesource.com/#/c/4720/

Imprecise tracking of parameter contents

Consider the following program:

```
func foo(p **int) *int {
        return *p
}

func main() {
    x := new(int) // BAD: new(int) escapes
    _ = foo(&x)
}
```

Currently new(int) incorrectly escapes, because flow through fake funcParam node is tracked incorrectly. It is easy to fix by introducing explicit flow from input parameters to results via fake indirection. The following change contains the fix:
https://go-review.googlesource.com/#/c/3886

Flow through fields

Example:

```
var x X
x.foo = &i
sink = x.bar // BAD: &i escapes
```

The problem here is that we don't track flow through individual fields, instead we coarsen it to whole structs. Proposed solution: attach a set of fake nodes representing fields to every struct ONAME. When we assign to a field, direct flow to the relevant fake node. All fake field nodes must also flow to the struct itself, because we can copy the whole struct. The fake fields can also be used for e.g. maps to represent all keys and all values, or for arrays to represent individual elements.
https://go-review.googlesource.com/#/c/3032/ contains a very rough prototype of the idea.

Retaking of string/byte slice storage

This is not exactly about escape analysis, but related.
From issue https://github.com/golang/go/issues/6714

```
        var b []byte
        b = append(b, "Hello, "...)
        b = append(b, "world.\n"...)
        return string(b) // BAD: allocates again
```

Compiler could figure out that string(b) is the last usage of b (b does not escape and is dead after this statement), and retake b storage to create the string. This can work in both directions: []byte(str) can also retake string storage, however in this case we need to ensure that the string is allocated on heap.

Read-only byte slices

This is not exactly about escape analysis, but related.
From https://github.com/golang/go/issues/2205

```
func isWS(b byte) bool {
        // BAD: allocate buffer for byte slice
        return bytes.IndexByte([]byte("\t\n\x0C\n "), b) != -1
}
```

Compiler could figure out that IndexByte uses the slice only for reading (most likely this implies that it does not escape), and then convert the string to byte slice without allocation and copying.

Append

The array allocated by append should be subject to escape analysis. If neither the slice nor the address of any element escapes, the array may be stack allocated, as in this example:
http://play.golang.org/p/uPR3d3i4fv

This suggests a further optimization unrelated to escape analysis: consolidation of a sequence of appends into a single allocation and initialization of an array of sufficient capacity.
http://play.golang.org/p/0dGi8fpY2Y
A number of append-related optimizations would be possible if it were broken down into its constituent operations (capacity check, reallocation) and then subjected to further analysis.

Litmus tests

There are two good real-world litmus tests for escape analysis. In the end we want the following code pattern to allocate buf on stack:

```
var buf bytes.Buffer
buf.Write([]byte{1})
_ = buf.Bytes()
```

And the following code pattern to allocate i on stack:

```
i := 0
sink = fmt.Sprint(&i)
```