Nir Boneh
Brian Kidd
Byron Samaripa
CSCI 3155

Adding Capacity Reduction in Google's GO Slice Syntax

Go, also commonly referred to as golang, is a programming language initially developed at Google in 2007 by Rob Pike, Robert Griesemer, and Ken Thompson. The language was announced in November 2009 and is now used in some of Google's production systems. Go's "gc" compiler targets the Linux, Mac OS X, FreeBSD,NetBSD, OpenBSD, Plan 9, and Microsoft Windows operating systems and the i386,amd64, and ARM processor architectures. Go's primary reason for development according to Rob Pike was to create and build a better C. It is a statically-typed language with things such as garbage collection, type safety, some dynamic-typing capabilities, and a large standard library, its syntax is loosely derived from C. One feature that the Go language supports like many other languages is slicing that was first introduced in Fortran 57. Google made some assumptions on what functions the programmer would need while interacting with slices and did not address all possible cases which inevitably lead to issue 1642. Before we can explain issue 1642 and the proposed solution that Russ Cox submitted to google, we must first explain how slices work in Go.

To understand what a slice is in Go, you must first understand what an array is in Go. Arrays in Go are not often used because the size of the array is part of its type. This is problem because it limits its expressive power. To declare an array in Go, it would look something like this: var buffer [256]byte. This will declare the variable buffer which can hold 256 bytes. The variable holds 256 bytes of data, and we can access its elements with the indexing syntax: buffer[0], buffer[1] all the way to buffer[255]. Attempting to index buffer with

a value outside this range will crash the program. Arrays have their place in Go, but their most common purpose is to hold storage for a slice.

Now that we have a better understanding of what an array is in Go, we can now move on to explain what a slice is. A slice is a data structure describing a contiguous section of an array stored separately from the slice variable itself. A slice is not an array, however; a slice describes a piece of an array. Given our previous example of the buffer array, we could now create a slice that describes elements 100 through 150 by slicing the array. This would be done by: var slice = buffer[100:150]. The result would slice the elements 100 through 150 but not include the 150th element. You can normally think of a slice being a little data structure with three elements: a length, a capacity, and a pointer to an array. For right now we will ignore the capacity since it will default to the length if it is not specified. So far we have only shown that you can slice an array, but that is not the extent to what you can slice. You can also create slices of a slice. This looks like: slice2 := slice[5:10]. just like before, this operation creates a new slice, but in this case with elements 5 (inclusive) through 10 (exclusive) of the original slice which means elements 105 through 109 of the original array. It's now time to talk about the third component of the slice: its capacity. The capacity field records how much space that the underlying array actually has, or the maximum value that the length can reach. Trying to grow the slice beyond its capacity will step beyond the limits of the array and cause problems. Normally when declaring a slice, the capacity will be equal to the length of the array minus the index in the array of the first element of the slice. Although that does not necessarily have to be the case, the capacity can be a lower value that does not reach the end of the array. This is however not supported well, Go lacked a way of

creating a new slice from an existing slice with a lower specified capacity and this lead to issue 1642.

While using Slice Syntax Rogge Pepper ran to a limitation with the language. Rogge Pepper ran into the problem that could have been fixed, if Go provider the ability to initialize a new slice with a lower capacity from an existing slice. The Issue was posted in the Google's Go community forum and named issue 1642. Here is a post on the Go community forum:

```
I was recently experimenting with an interface to allow
data to be moved between processing modules:
type BlockReader interface {
    ReadBlock() ([]byte, os.Error)
}
type BlockWriter interface {
    WriteBlock([]byte) os.Error
}
Writing a block of data hands control of the
data to the receiver; reading a block hands control of it
to the caller (one implementation would be to use a chan []byte).
It would be useful if the controller of a block
could append to a given block that had sufficient capacity without
copying it.
However that's not safe, as we'd like
to allow a writer to carve up a block and send
its component pieces in independent BlockWrites:
e.g.
   func Splitter(b BlockWriter, data []byte, size int) {
```

```
    for len(data) > size {
        b.WriteBlock(data[0:size])
        data = data[size:]
    }
    b.WriteBlock(data)
}
```

There's no way of handing out a slice of data without
implicitly handing out all the rest of data too.
The alternative is that every time a block is extended,
it is copied.
This is always an issue when handing over
control of a slice to somewhere else - the only
way to be sure currently is to copy it, which is often overkill.
How about we allow slicing the capacity of a slice as well
as its length. It could be an optional 3rd expression in a slice
expression x[start:len:capacity], e.g.

```
    b.WriteBlock(data[0:size:size])
```

I realise that this breaks the current assumption that
&a[cap(a)-1] == &b[cap(b)-1] implies a and b point to the same
slice, but is that actually important?

The main point to take away from this quote is that he could not achieve his desired goal with the current implementation of Go. He could technically achieve his desire goal of lowering the capacity of the slice by making a completely new copy of the slice, but that it is far too much of an overkill for what he is trying to achieve and takes more computational power than actually necessary.

Nir Boneh
Brian Kidd
Byron Samaripa
CSCI 3155

A proposed solution was later introduced by Russ Cox in June 2013. He proposed to add to Go 1.2 a variant of the slice operation that allows reducing the capacity of a slice. There are some cases it would be useful, for example in custom []byte allocation managers, to be able to hand a slice to a caller and know that the caller cannot edit values beyond a given subrange of the true array. The addition of append to the language made this somewhat more important, because append lets programmers overwrite entries between len and cap without realizing it or even mentioning cap. Although issues such as issue 1642 don't come up often, they do come up occasionally. The most important argument in favor is that giving programmers a way to control cap gives them more control over append.

The proposed syntax to accomplish the task of creating another slice from an existing slice with a designated lower capacity is the following a[i : j : k]. The result has indices starting at 0, length equal to j - i, and capacity equal to k - i. The evaluation panics if $i \leq j \leq k \leq cap(a)$ is not true. For convenience, the index i may be omitted: it defaults to zero. The other indices j and k are required. Once we have sufficient experience using this form we might choose appropriate defaults for them. This does not remove the syntax of a[i:j] which means the proposed solution is backward compatible and most programmers can choose to ignore it. This addresses an issue that came in with the Google GO's community which stated that the syntax itself is confusing. Since the addition of this syntax is optional for use and the old syntax is still valid, the issue is invalid. The most important implication is that control over cap gives programmers control over append, especially when handing a slice to code in another package.

Nir Boneh
Brian Kidd
Byron Samaripa
CSCI 3155

One problem is due to this new feature, the ability to check for aliasing using: &x[:cap(x)][cap(x)-1] == &y[:cap(y)][cap(y)-1] is now unsafe. This is something Rob Pike specifically stated before considering to implement this feature. Russ Cox addressed this by stating that in practice this test is not widely used. The only code he can find anywhere (including public Go code) containing such a test is math/big's func alias. Since math/big already uses assembly, it would be reasonable for it to use unsafe to do a real pointer comparison instead. There is simply not a compelling argument that overlap testing is important enough to warrant language support beyond package unsafe.

The last issue is deciding the default values j and k should have incase the programmer does not specify those two values. There are a number of sensible possibilities for what the defaults for j and k should be. Russ Cox addresses all the possible cases for when the programmer does not specify the j or k and proposed a sensible solution for each:

Case 1: j defaults to len(x), k defaults to cap(x).

x[i::] ≡ x[i:]

x[i:j:] ≡ x[i:j]

x[::] ≡ x

x[::n] depends on n vs len

x[::15] ≡ x[0:10:15]

x[::5] panics (implicit j = 10 > k = 5)

x[0:15:] ≡ x[0:15]

Case 2: j defaults to k, k defaults to cap(x)

x[i::] ≡ x[i:20]

x[i:j:] ≡ x[i:j]

x[::] ≡ x[:20] // an idiom for growing a slice?

x[::n] ≡ x[0:n:n]

x[::15] ≡ x[0:15:15]

x[::5] ≡ x[0:5:5]

x[0:15:] ≡ x[0:15]

Case 3: j defaults to min(len(x), k), k defaults to cap(x)

x[i::] ≡ x[i:]

x[i:j:] ≡ x[i:j]

x[::] ≡ x

x[::n] depends on n vs len

x[::15] = x[0:10:15] // len stayed where it was

x[::5] ≡ x[0:5:5] // len pulled down by new cap

x[0:15:] ≡ x[0:15]

Case 4: j defaults to len(x), k defaults to len(x)

x[i::] ≡ x[i:10:10], NOT same as x[i:]

x[i:j:] ≡ x[i:j:10], NOT same as x[i:j]

x[::] ≡ x[0:10:10] // an idiom for shrinking a slice?

x[::15] ≡ x[0:10:15]

x[::5] panics (implicit j = 10 > k = 5)

x[0:15:] panics (j = 15 > implicit k = 10), NOT same as x[0:15]

Case 5: j defaults to k, k defaults to len(x).

x[i::] ≡ x[i:10:10], NOT same as x[i:]

x[i:j:] ≡ x[i:j:10], NOT same as x[i:j]

x[::] ≡ x[:10:10] // an idiom for shrinking a slice?

x[::15] ≡ x[0:15:15] // slice cap shrank, len grew

x[::5] ≡ x[0:5:5]

x[0:15:] panics (j = 15 > implicit k = 10)

Case 6: j defaults to min(len(x), k), k defaults to len(x).

x[i::] ≡ x[i:10:10], NOT same as x[i:]

x[i:j:] ≡ x[i:j:10], NOT same as x[i:j]

x[::] ≡ x[:10:10] // an idiom for shrinking a slice?

x[::15] ≡ x[0:10:15] // len stayed where it was

x[::5] ≡ x[0:5:5] // len pulled down by new cap

x[0:15:] panics (j = 15 > implicit k = 10)

Without this syntax the GO language is taking away power from the programmer that he otherwise should have. The GO language just doesn't feel complete without it. The proposed solution is backwards compatible and does not force programmers to use it, so most programmers should not be affected by the addition. In the rare case such as issue 1642 where the feature is needed, the GO language should simply offer this simple solution syntax to the programmer without driving him nuts. When designing a language one should always assume that the programmer might need the power to performs all possible cases, even if it does not seem useful. This is why Russ Cox solution was approved and implemented in GO 1.2