opCode

VIRTUAL MACHINE EXAMPLES

ANTHONY DAVIS

free chapter preview

opCode: virtual machine examples, by Anthony Davis
opcodebook.com

free chapter preview

This preview is provided free of charge and may not be sold.

PDF preview version 20191213

about

We start with a working virtual machine (VM) and explain the use and building of opcodes for it. I'll explain how I went about adding some very useful features to a language, such as decoupling assignment, multi-variable assignment, short-circuiting operators, string interpolation, *for* loops, scoped *if* expressions, and exceptions. We'll review how opcodes make these things work in a VM.

This is not about CPU's and their instructions. It also does not explain parsing or cover the langur parser. I assume some programming experience. I also assume you know or can easily pick up some Go language coding without having to explain all of it, though I do touch on some of Go's features.

The working language we start with is langur, a scripting language I built with Go (also see acknowledgments). To make the best use of this book, you'll need to download and install langur, or at least view the source code. (There is much source code not shown in this book.) Instructions for installation are included on langurlang.org and in the download.

This version of this book is based on langur 0.5 beta. The code you download may look different than what you see here. The website langurlang.org explains the features of langur. Also, the site rosettacode.org has some code samples.

These opcodes are not intended to diagnose, treat, prevent, or cure any disease.

formatting conventions used

Besides Go code sample blocks, which use a monospace font and multiple colors, the following formatting is used within this book.

`langur source code`
`Go and other source code`
`REPL text`
language structure
opcodes section

files

I'll often refer to files in a langur package with a short path. For example, instead of src/langur/object/object.go, I'll simply use object/object.go.

acknowledgments

Thanks to those who were patient enough with me to allow me to write and publish a book.

Thanks to God for everything good.

I first wrote langur following very useful books by Thorsten Ball (*Writing an Interpreter in Go* and *Writing a Compiler in Go*), but the code is sometimes very different from Monkey (the name of his language).

The font Bitwise (www.1001freefonts.com/bitwise.font) is attributed to Digital Graphic Labs.

The font DATA CONTROL (www.1001freefonts.com/data-control.font) is attributed to Vic Fieger (www.vicfieger.com/).

contents

7

retrieving non-constant values

Besides constants, there are a number of other things to specifically push onto the stack. These include global variables, local variables, non-local variables, free variables, and self references.

A local variable is one referenced in the same frame in which it is stored. A non-local is a reference to a variable stored in a different frame than it is being referenced from (not including global variables).

Free variables are discussed later when we talk about function closures, and self references are used for direct recursion of functions.

value retrieval in the VM

To retrieve things onto the stack, we use OpGetGlobal, OpGetLocal, OpGetNonLocal, OpGetFree, and OpGetSelf. We can find these and their operand widths in opcode/opcode.go.

```
OpGetGlobal:   {"GetGlobal", []int{2}},
OpGetLocal:    {"GetLocal", []int{1}},
OpGetNonLocal: {"GetNonLocal", []int{1, 1}},
OpGetBuiltIn:  {"GetBuiltIn", []int{2}},
OpGetFree:     {"GetFree", []int{1}},
OpGetSelf:     {"GetSelf", []int{}},
```

In the VM instruction loop, we find the following cases.

```
case opcode.OpGetGlobal:
    globalIndex := opcode.ReadUInt16(ins[ip+1:])
    ip += 2
    err = vm.push(vm.globalFrame.locals[globalIndex])

case opcode.OpGetLocal:
    localIndex := int(ins[ip+1])
    ip += 1
    result, err = fr.getLocal(localIndex)
    if err == nil {
        err = vm.push(result)
    }
```

```
case opcode.OpGetNonLocal:
    index := int(ins[ip+1])
    level := int(ins[ip+2])
    ip += 2

    result, err = fr.getNonLocal(index, level)
    if err == nil {
        err = vm.push(result)
    }
```

The instruction pointer (`ip` variable) is pointing to the opcode, so we use `ip+1`, `ip+2`, etc. to read in the operands. The function call `opcode.ReadUInt16(ins[ip+1:])` returns an integer, reading in the first 2 bytes from the slice it is passed (the colon being a range operator in the context of `ins[ip+1:]`). To read a single byte operand, we just read it directly (such as `int(ins[ip+2])`), using a type cast (`int(...)`) to convert it from a byte to an integer.

Having read in the operands, we advance the instruction pointer the same width as the operands (`ip += 2`). We don't advance it past the operands (will point at the last byte of the last operand), as it will be automatically advanced by 1 byte on each iteration of the instruction loop.

OpGetGlobal and OpGetLocal each have one operand, but OpGetNonLocal has two, an index and a level (or you might say frame distance).

For local value retrieval, we have the following function (in vm/frames.go).

```
func (fr *frame) getLocal(localIndex int) (
    obj object.Object, err error) {

    if fr.code.LocalBindingsCount > 0 {
        return fr.locals[localIndex], nil
    }
    return fr.base.getLocal(localIndex)
}
```

The check `fr.code.LocalBindingsCount > 0` essentially checks to see if the current frame has scope. If not, the `getLocal()` must be asking for a value in another frame.

For non-local value retrieval, we have the following.

```
func (fr *frame) getNonLocal(localIndex, count int) (
    obj object.Object, err error) {

    if count == 0 {
        return fr.locals[localIndex], nil
    }
    return fr.base.getNonLocal(localIndex, count-1)
}
```

We check the count and if not 0, decrement the count, checking the next base frame.

With getNonLocal, could we have eliminated “globals” in langur? I did not, so far, as it seemed good to keep them for at least a couple of reasons.

1. No matter how many frame levels deep you are, globals can be easily accessed by addressing them separately. This might be more efficient?
2. I’ve used a single byte index for locals, but a 2-byte index for globals. This means locals are limited to 256 values, but globals could include much more than that.

indexed value retrieval in the VM

Indexed retrieval is used when you have something like the following.

```
var .x = [7, 21, 35]
.x[3]
# result == 35
```

An alternate value may be used to return instead of an exception for an invalid index, such as in the following. Short-circuiting is used for the alternate value (not evaluated if not used).

```
var .x = [7, 21, 35]
.x[3; 123]    # 35
.x[7; 123]    # 123
```

The VM has the following 2 cases in its instruction loop, which call `executeIndexOperation()`.

```
case opcode.OpIndex:
    _, err = vm.executeIndexOperation(fr, false, 0)

case opcode.OpIndexAlternate:
    shortCircuitJump := int(opcode.ReadUInt16(ins[ip+1:]))
    ip += 2

    var jumpAlt bool
    jumpAlt, err = vm.executeIndexOperation(fr, true, shortCircuitJump)
    if jumpAlt {
        ip += shortCircuitJump
    }
```

The beginning of `executeIndexOperation()` may appear a little odd, since I said that it would use short-circuiting for an alternate value. The reality is that if the alternate is very simple, such as a number or a string without interpolation, it might not use short-circuiting evaluation on it, as it could be a waste. So, if there is an alternate, but no short-circuiting, we must first pop that off the stack as it has already been evaluated.

```
func (vm *VM) executeIndexOperation(
    fr *frame, alt bool, shortCircuitJump int) (
    jumpAlt bool, err error) {

    var alternate object.Object // nil by default
    if alt && shortCircuitJump == 0 {
        // have alternate with no short-circuiting
        alternate = vm.pop()
    }
```

Then, as we'll see in discussing index opcodes and compiling them, we first pushed the indexable value to the stack ("`left`" below), then the `index`. We must pop them off the stack in the opposite order.

```
    index := vm.pop()
    left := vm.pop()
```

Then, we evaluate these things to get a result or an error. If `alternate` is not `nil` (already known), then `object.Index()` will use that to return instead of an error for an invalid index. It would also return false for `useAlternate` since the result would already be known.

```
    result, useAlternate, err := object.Index(left, index, alternate)
    if err != nil {
        if alt && useAlternate {
            return false, nil
        }
        return true, err
    }

    return true, vm.push(result)
}
```

If we received an index error and have an alternate value yet to evaluate and use and `object.Index()` indicates that we should use the alternate, we then use `return false, nil` to return to the instruction loop, `false` meaning don't jump over an alternate and `nil` as we are not returning an error to the instruction loop. If we received another kind of error (unlikely), we use `return true, err` to return to the instruction loop with an error, which the instruction loop will convert to a langur exception (see the "exceptions" chapter).

If all is well, we return `true` and any error returned from `vm.push()` (likely `nil`, not an error).

The `object.Index()` method and the functions it calls don't fit into this short book, but you can view them in the object/index.go file to see how this works.

global / local value retrieval opcodes

If we declare and set a variable `.x` (see "declaration and assignment"), then simply type `.x` in the REPL, we can see an OpGetGlobal which retrieves the value.

```
>> val .x = 21
ByteCode Instructions
0000 Constant 1
0003 SetGlobal 4
0006 Pop

>> .x
ByteCode Instructions
0000 GetGlobal 4
0003 Pop

langur escaped result: 21
```

The `GetGlobal 4` retrieves a value set by `SetGlobal 4`. The 4 is an index into the global variables slice. Once established, an index does not change, so that a variable value may be retrieved or changed by another opcode at any point thereafter. The compiler, with its symbol tables, keeps track of indices to produce opcodes to set and retrieve a given variable by name (as discussed in the chapter on "declaration and assignment"). The VM never sees a variable name (just a number).

Opcodes for retrieving local values (those stored in the same frame as the instructions to retrieve them) look the same as for retrieving global values, except that they use OpGetLocal instead of OpGetGlobal.

non-local value retrieval opcodes

So, what's a non-local? It's not a foreigner, but a value stored on a frame higher than the frame executing the instruction to retrieve it. This being the case, an OpGetNonLocal requires another operand, the frame distance, or level, to go to find the value.

Type the following in the REPL. We have 2 scoped blocks of code. The inside block retrieves a value from the outside block.

```
>> { val .x = 7; { val .y = .x x 3 } }
ByteCode Instructions
0000 Execute 4
0003 Pop

ByteCode Constants
...
3: Code (...); LocalBindingsCount: 1
Instructions
0000 GetNonLocal 0 1
0003 Constant 2
0006 Multiply
0007 SetLocal 0

4: Code (...); LocalBindingsCount: 1
```

```
Instructions
0000 Constant 1
0003 SetLocal 0
0005 Pop
0006 Execute 3

langur escaped result: 21
```

In this example, `0000 Execute 4` executes the code in constant 4. This is the outside block of code where `.x` is set to `7`. Note that the index used is 0 (`0003 SetLocal 0`). It then has instruction `0006 Execute 3` to execute the code in constant 3.

In constant 3, we have `0000 GetNonLocal 0 1`, with 2 operands, an index value and a frame level. The 0 indicates the index on the distant frame and the 1 indicates that it is 1 frame away. In that other frame, we had set `.x` using index 0, so we use the same index to retrieve it. We don't use OpGetLocal since we're in a different scoped frame, with its own local values.

The `0007 SetLocal 0` sets the value of `.y` in the inner block of code. There's no conflict between this local index 0 and the other local index 0 since they are in different frames.

indexed value retrieval opcodes

Langur allows indexing arrays, strings, and hashes by single items or by arrays. Indexing by array returns an array. Arrays and strings can also be indexed by range, or ranges may be used with indexing by array. All of the following are potentially valid for indexed value retrieval.

```
.x[1]
.x[7; 42]    # with index alternate 42
.x[2..7]
.x[[3, 4..6, 9..1]]
.y["abc"]
.y[["abc", "def"]]
.z[.x]
.z[.x[3] to 42]
```

For simplicity, and since it has no bearing on the other things we'll be looking at, we'll use a single index.

First define an indexable value in the REPL. Then, use square brackets to retrieve an indexed value from it. Note that langur uses 1-based indexing.

```
>> val .x = [4, 8, 9, 10]
    ...
>> .x[3]
ByteCode Instructions
0000 GetGlobal 4
0003 Constant 5
0006 Index
0007 Pop

ByteCode Constants
1: Number 4
2: Number 8
```

```
3: Number 9
4: Number 10
5: Number 3

langur escaped result: 9
```

In the retrieval process, we see `0000 GetGlobal 4`, which pushes the array represented by `.x` onto the stack. Then, we have `0003 Constant 5`. Looking at our constants, we see that this pushes the number `3` to the top of the stack. Then, `0006 Index` tells the VM to pop the index and thing to be indexed off the stack, and push the result, which in this case is the number `9`, the third element of array `.x`.

While we do have the number 9 in the constants slice, that is not where the result is coming from. That 9 is from creating the original array. I suppose the REPL could do some house-keeping to eliminate things it isn't using anymore, and it might at some point, and then you wouldn't see the 9 in the constants slice as in the example.

Using an index alternate looks like the following.

```
>> .x[5; 100 x 8]
ByteCode Instructions
0000 GetGlobal 4
0003 Constant 6
0006 IndexAlternate 7
0009 Constant 7
0012 Constant 2
0015 Multiply
0016 Pop

langur escaped result: 800
```

I've spared us from viewing the constants again. The `0000 GetGlobal 4` again retrieves the value of `.x` (the array set earlier) to the stack. `0003 Constant 6` pushes the number `5` onto the stack. But then, instead of OpIndex, we have OpIndexAlternate. This includes the number of bytes to jump if an index is valid (over the alternate). In this case, it would jump 7 bytes from `0009` to `0016`, short-circuiting evaluation of the alternate value.

compiling non-constant value retrieval

Symbol tables will be discussed when we talk about assignment, but I'll mention them now. We use symbol tables to keep track of variables in the compiler, and "push" and "pop" symbol tables in the compiler as required for scope.

To know if we've defined a variable, we check for it by calling `c.symbolTable.resolve()` as shown in `compileVariableNode()` below. It will return the symbol if it can be found. The symbol contains an index number (reserved slot) into the appropriate slice of values. This index number is then used to compile both value setting and retrieval so that all the VM sees is an index number.

```
func (c *Compiler) compileVariableNode(node *ast.VariableNode) (
    ins opcode.Instructions, err error) {

    sym, cnt, ok := c.symbolTable.resolve(node.Name)
    if !ok {
        err = makeErr(node, fmt.Sprintf(
            "Undefined variable .%s", node.Name))
        return
    }

    ins, err = c.makeOpGetInstructions(node, sym, cnt)
    return
}
```

Given a defined symbol, `makeOpGetInstructions()` helps us generate the right opcodes for each symbol type.

```
func (c *Compiler) makeOpGetInstructions(
    node ast.Node, sym symbol, level int (
    ins opcode.Instructions, err error) {

    switch sym.Scope {
    case globalScope:
        return opcode.Make(opcode.OpGetGlobal, sym.Index), nil
    case localScope:
        if level == 0 {
            return opcode.Make(opcode.OpGetLocal, sym.Index), nil
        }
        return opcode.Make(opcode.OpGetNonLocal, sym.Index, level), nil
    case freeScope:
        return opcode.Make(opcode.OpGetFree, sym.Index), nil
    case selfScope:
        return c.compileSelfRef(node)
    }
    err = makeErr(node, fmt.Sprintf("Attempt to create OpGet on .%s for %s",
        sym.Name, sym.Scope))
    bug("makeOpGetInstructions", err.Error())
    return nil, err
}

func (c *Compiler) compileSelfRef(node ast.Node) (
    opcode.Instructions, error) {

    if c.symbolTable.Outer == nil {
        return nil, makeErr(node, "Cannot use self token in global scope")
    }
    return opcode.Make(opcode.OpGetSelf), nil
}
```

These call `opcode.Make()`, which we give the exclusive right to build opcodes to avoid some potential errors in doing so.

compiling indexed value retrieval

For indexed value retrieval, we have `compileIndexExpression()`, which takes an `*ast.IndexNode` and turns it into opcodes.

```
func (c *Compiler) compileIndexExpression(node *ast.IndexNode) (
    ins opcode.Instructions, err error) {

    var b []byte

    // Get "left" node
    b, err = c.compileNode(node.Left, true)
    if err != nil {
        return
    }
    ins = append(ins, b...)

    // Get the index
    b, err = c.compileNode(node.Index, true)
    if err != nil {
        return
    }
    ins = append(ins, b...)
```

First, we compile the thing to be indexed (the “left” operand) and the index. Pretty straightforward. The alternate index value can make it more complicated.

```
    if node.Alternate == nil {
        ins = append(ins, opcode.Make(opcode.OpIndex)...)
```

If there is no alternate, this is sufficient and will be returned as the full instruction set. That is, we have instructions to generate or retrieve the thing to be indexed, then the index itself, then an OpIndex. In the VM, with our 2 items at the top of the stack, as discussed earlier, OpIndex will tell it to pop them off and do the indexing.

If there is an alternate, we have to generate more instructions. First, we compile the alternate without adding it to the instruction set (variable `ins`), as we don’t yet know where it goes.

```
    } else {
        // alternate for an invalid index
        var alt opcode.Instructions
        alt, err = c.compileNode(node.Alternate, true)
        if err != nil {
            return
        }
```

Having done this, we ask if short-circuiting is necessary. It will not be used for some simple cases.

```
        useShortCircuit := true
        switch alt := node.Alternate.(type) {
        case *ast.NumberNode, *ast.NullNode, *ast.BooleanNode:
            // no short-circuiting on some simple things
            useShortCircuit = false
        case *ast.StringNode:
            // short circuit for strings that include interpolation
            useShortCircuit = len(alt.Interpolations) > 0
        }

        if useShortCircuit {
            ins = append(ins, opcode.Make(
                opcode.OpIndexAlternate, len(alt))...)
            ins = append(ins, alt...)
        } else {
            ins = append(ins, alt...)
            ins = append(ins, opcode.Make(opcode.OpIndexAlternate, 0)...)
        }
    }

    return
}
```

And finally, if using short-circuiting, we include the length to jump as an operand of OpIndexAlternate. If not, we use 0. Also, you'll notice that with short-circuiting, the alternate instructions are placed after the OpIndexAlternate, and without short-circuiting, they are placed before it.

Then, we return with the instructions we've built. That is all there is to compiling indexed value retrieval in langur. Setting indexed values is an entirely different process for langur.

preview fini

I hope you've found this preview informative. Visit opcodebook.com for information about or to purchase the book.

Langur is open source, free software. Visit langurlang.org to see about the language.

book chapters