Technische Universität Berlin
Compiler Construction and Programming Languages
Prof. Dr. Peter Pepper · Secr. TEL12-2 · www.uebb.tu-berlin.de

Compiler Construction 1
Pepper · Lorenzen

Winter term 2014/2015
Handout: Jan 22, 2015
Submission: Feb 4, 2015

Milestone 4

Code generator for µ-Opal

Assignment Code generator
Implement a code generator that translates a µ-Opal program into a sequence of COVM instructions. The values computed by the COVM code and your interpreter of milestone 3 must be identical. Make sure that your generated code aborts on a `sub` underflow. This has to be handled by the code generator since µ-Opal has natural numbers (non-negative integers) but the COVM supports also negative integers. Your code generator must return code for every context correct input program, that is, it must not abort with an error.[1]
The package `de.tuberlin.uebb.comp1.covm.instructions` contains the definitions of the COVM instructions and is included in the µ-Opal compiler template.
Submit the source tree of your µ-Opal compiler implementation including your implementation of the code generator.

File to modify: `CodeGenerator.scala`

The Compiler Construction 1 Virtual Machine (COVM)
(Repeated from milestone 1a.)

General Notes The COVM (Compiler Construction 1 Virtual Machine) is a simple stack machine. Its primitive data types are integers and tuples and it supports conditional branches and function calls. The value stack is an empty or non-empty sequence of words:

ws	::=	[]	empty stack
		$w : ws$	non-empty stack
w	::=	n	integers
		a	addresses
		$\langle w, ..., w \rangle$	tuples

A word is either an integer n, an address of the code segment a (a non-negative integer), or a tuple of words $\langle w_0, ..., w_m \rangle$.
A machine state is a triple of a non-negative integer ip (the instruction pointer), a code segment C and a value stack ws:

$$state \subseteq ip \times C \times ws$$

A stack machine programm is an instruction sequence where the first element is the start instruction. The initial state for a code segment $instr_0 \cdots instr_m$ is

$$(0, instr_0 \cdots instr_m, []).$$

A successful final state of machine has a stack of size one. A failure during execution will stop the machine immediately.

[1] The result type of the code generator is `Either[Diag,List[Instruction]]` but the sole purpose of `Either` is to indicate that the code generator is not yet implemented in previous milestones.

Instructions

The COVM has 19 instructions. Each instruction is described in prose and by the corresponding state transition $state_1 \rightarrow state_2$.
By the notation $C_k[instr]$ for a state (ip, C, ws) we indicate that *instr* is the k-th instruction in the code segment C. For example, $(ip, C_{ip}[call], ws)$ means that the instruction pointer references a *call* instruction. In the COVM, the instruction pointer always points to the instruction that will be executed in the next step.

Arithmetic instructions

The arithmetic instruction do not influence the linear control flow. Therefore, the instruction pointer is always incremented by one to continue with the next instruction.
All arithmetic instructions expect two integers on top of the stack. This is indicated in the state transitions by the variable names starting with n which stands for integer words.

add Add the first two stack words.

$$(ip, C_{ip}[add], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 + n_0 : ws)$$

The machine aborts if there is an integer overflow.

sub Subtract the first word from the second word of the stack.

$$(ip, C_{ip}[sub], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 - n_0 : ws)$$

The machine aborts if there is an integer underflow.

mul Multiply the first two words of the stack.

$$(ip, C_{ip}[mul], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 \cdot n_0 : ws)$$

The machine aborts if there is an integer overflow.

div Divide the second word by the first word of the stack.

$$(ip, C_{ip}[div], n_0 : n_1 : ws) \rightarrow (ip + 1, C, n_1 \div n_0 : ws)$$

The machine aborts if there is a division by zero.

Push and pop stack words

Similar to the arithmetic instructions all of the following instructions do not modify the linear control flow.

pushint n Push an integer constant n onto the stack.

$$(ip, C_{ip}[pushint\ n], ws) \rightarrow (ip + 1, C, n : ws)$$

pushaddr a Push an address of an instruction of the code segment onto the stack.

$$(ip, C_{ip}[pushaddr\ a], ws) \rightarrow (ip + 1, C, a : ws)$$

push i Copy the i-th stack word to the top of the stack.

$$(ip, C_{ip}[push\ i], w_0 : \cdots : w_i : ws) \rightarrow (ip + 1, C, w_i : w_0 : \cdots : w_i : ws)$$

slide i Remove i words $w_1, ..., w_i$ from the top of the stack but keep the first one w_0. This instruction is handy to remove arguments to a function from the stack.

$$(ip, C_{ip}[slide\ i], w_0 : w_1 : \cdots : w_i : ws) \rightarrow (ip + 1, C, w_0 : ws)$$

swap Swap the first two words of the stack.

$$(ip, C_{ip}[swap], w_0 : w_1 : ws) \rightarrow (ip + 1, C, w_1 : w_0 : ws)$$

pack i Pack i values into a tuple.

$$(ip, C_{ip}[pack\ i], w_0 : \cdots : w_{i-1} : ws) \rightarrow (ip + 1, C, \langle w_{i-1}, ..., w_0 \rangle : ws)$$

unpack i Push the i-th component of a tuple onto the stack.

$$(ip, C_{ip}[unpack\ i], \langle w_0, ..., w_i, ..., w_m \rangle : ws) \rightarrow (ip + 1, C, w_i : ws)$$

Control flow instructions

Control flow instruction like jumps and function calls and returns set the instruction pointer in various ways.

call Call the function the top word a of the stack points to (by setting the instruction pointer to a) and saves the return address on the stack. The return address is the address of the instruction following the *call*, that is, $ip + 1$.

$$(ip, C_{ip}[call], a : ws) \rightarrow (a, C, ip + 1 : ws)$$

ret Return from a function call by setting the instruction pointer to the return address which is the second word on the stack. The result of the function w (the first word of the stack) is kept.

$$(ip, C_{ip}[ret], w : a : ws) \rightarrow (a, C, w : ws)$$

jmp a Jump to the absolute address a.

$$(ip, C_{ip}[jmp\ a], ws) \rightarrow (a, C, ws)$$

jz a Jump to the absolute address a if the top level word is zero. Continue with the next instruction if this word is not zero. The top-most word of the stack is removed in either case.

$$\begin{aligned} &(ip, C_{ip}[jz\ a], 0 : ws) \rightarrow (a, C, ws) \\ &(ip, C_{ip}[jz\ a], n : ws) \rightarrow (ip + 1, C, ws) && \text{if } n \neq 0 \end{aligned}$$

jlt a Jump to the address a if the top level word is less than zero. Continue with the next instruction if this word is greater than or equal to zero. The top-most word of the stack is removed in either case.

$$\begin{aligned} &(ip, C_{ip}[jlt\ a], n : ws) \rightarrow (a, C, ws) && \text{if } n < 0 \\ &(ip, C_{ip}[jlt\ a], n : ws) \rightarrow (ip + 1, C, ws) && \text{if } n \geq 0 \end{aligned}$$

jgt a Jump to address a if the top level word is greater than zero. Continue with the next instruction if this word is less than or equal to zero. The top-most word of the stack is removed in either case.

$$\begin{aligned} &(ip, C_{ip}[jgt\ a], n : ws) \rightarrow (a, C, ws) && \text{if } n > 0 \\ &(ip, C_{ip}[jgt\ a], n : ws) \rightarrow (ip + 1, C, ws) && \text{if } n \leq 0 \end{aligned}$$

stop Stop the machine. The current state must be a valid final state.

abort s Abort execution with an error message s.

A small example

Consider this small µ-Opal program:

```
DEF MAIN: nat == foo(1, 5, 3)
DEF foo(x: nat, y: nat, z: nat): nat == add(x, sub(y, z))
```

A possible translation of this program to COVM instructions is the following code. Each instruction is annotated with the stack that is active before the instruction is executed, that is, *ip* points to that instruction. Addresses on the stack are prefixed with `@` to avoid confusion with integers.

```
--  current instruction   stack
--  Code of MAIN
00  pushint 1             -- stack: []
01  pushint 5             -- stack: 1 : []
02  pushint 3             -- stack: 5 : 1 : []
03  pushaddr 7            -- stack: 3 : 5 : 1 : []
04  call                  -- stack: @7 : 3 : 5 : 1 []
05  slide 3               -- stack: 3 : 3 : 5 : 1 : []
06  stop                  -- stack: 3 : []

--  Code of foo
07  push 3                -- stack: @5 : 3 : 5 : 1 : []
08  push 3                -- stack: 1 : @5 : 3 : 5 : 1 : []
09  push 3                -- stack: 5 : 1 : @5 : 3 : 5 : 1 : []
10  sub                   -- stack: 3 : 5 : 1 : @5 : 3 : 5 : 1 : []
11  add                   -- stack: 2 : 1 : @5 : 3 : 5 : 1 : []
12  ret                   -- stack: 3 : @5 : 3 : 5 : 1 : []
```

The three arguments for the function `foo` are pushed onto the stack in instructions `00` to `02`. The function `foo` starts at address `07` in the code segment. In order to call `foo`, its address `@7` is pushed onto the stack in instruction `03`. The call in instruction `04` jumps to this address and pushes the return address `@5` onto the stack. After the call, the three arguments for `foo` are popped from the stack by the *slide* instruction in `05`. The machine stops with a valid final configuration in instruction `06`.
The function `foo` first pushes `x` onto the stack. `x` is word number 3 of the stack as the return address is the top-most word. Then the arguments `y` and `z` for `sub` are pushed onto the stack. Since the stack grows while pushing arguments, the position of each argument happens to be 3. After execution of `sub` and `add`, the `ret` instruction jumps back to the return address `@5` and removes it from the stack.

The COVM implementation

The COVM implementation is contained in the µ-Opal compiler template and started by the compiler to execute the generated code. However, it is also possible to run the COVM stand-alone, similar to the Java Virtual Machine implementation `java`.
The ISIS page provides the source distribution of the COVM. You can compile is using SBT and create a start script with the `start-script` SBT command.
The COVM takes a text file containing a sequence of instructions as argument. For example, let `foo.S` be the code of the previous section:

```
$ target/start foo.S
Result: 3
```

If you encounter a `Stack overflow` or an `Out of heap memory`[2] exception you can increase the respective values with the command line switches `-h` and `-s`. The default values are 4096 words for the

[2]You might be surprised to read of a heap since COVM was announced as a stack machine. It is a stack machine as opposed to a register machine but it needs a heap to store the actual values. All the details will be discussed in the memory management chapter of the lecture.

heap and 512 for the stack.
The COVM implementation offers a trace mode that shows the current instruction pointer (and its target) and the content of the stack. It is enabled by the `-d trace` flag and produces an output similar to the comments in the listing of the previous section.

```
$ target/start -d trace foo.S
ip = 0     { pushint 1 }      []
ip = 1     { pushint 5 }      1 : []
ip = 2     { pushint 3 }      5 : 1 : []
ip = 3     { pushaddr 7 }     3 : 5 : 1 : []
ip = 4     { call }           @7 : 3 : 5 : 1 : []
ip = 7     { push 3 }         @5 : 3 : 5 : 1 : []
ip = 8     { push 3 }         1 : @5 : 3 : 5 : 1 : []
ip = 9     { push 3 }         5 : 1 : @5 : 3 : 5 : 1 : []
ip = 10    { sub }            3 : 5 : 1 : @5 : 3 : 5 : 1 : []
ip = 11    { add }            2 : 1 : @5 : 3 : 5 : 1 : []
ip = 12    { ret }            3 : @5 : 3 : 5 : 1 : []
ip = 5     { slide 3 }        3 : 3 : 5 : 1 : []
ip = 6     { stop }           3 : []
Result: 3
```