CS143 Spring 2022 – Written Assignment 4

Tuesday, May 31, 2022 11:59 PM PDT

This assignment covers code generation, operational semantics, and optimization. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by Tuesday, May 31, 2022 at 11:59 PM PDT. A LaTeX template for writing your solutions is available on the course website.

1. Consider the following program in Cool, representing a "slightly" over-engineered implementation which calculates the sum of $\{0, 1, 2\}$ using an operator class and a reduce() method:

```
class BinOp {
    optype(): String {
        "BinOp"
    };

    operate(a: Int, b: Int): Int {
        a + b
    };
};

class SumOp inherits BinOp {
    optype(): String {
        "SumOp"
    };
};

class MulOp inherits BinOp {
    optype(): String {
        "MulOp"
    };

    operate(a: Int, b: Int): Int {
        a * b
    };
};

class IntList {
    head: Int;
    tail: IntList;
    empty_tail: IntList; -- Do not assign.

    tail_is_empty(): Bool {
        tail = empty_tail
    };

    get_head(): Int { head };

    set_head(n: Int): Int {
        head <- n
    };

    get_tail(): IntList { tail };

    set_tail(t: IntList): IntList {
        tail <- t
    };

    generate(n: Int): IntList {
        let l: IntList <- New IntList in {
            l.set_head(n);
            -- Point A
            if (n = 1) then
                l.set_tail(empty_tail)
            else
                l.set_tail(generate(n-1))
            fi;
            l;
        }
    };
};

class Main {
    reduce(result: Int, op: BinOp, l: IntList): Int {{
        result <- op.operate(result,l.get_head());
        if (l.tail_is_empty() = true) then
            -- Point B
            result
        else
            reduce(result,op,l.get_tail())
        fi;
    }};

    main(): Object {
        let op: BinOp <- new SumOp, l: IntList <- new IntList,
            io: IO <- new IO in {
            l <- l.generate(2);
            io.out_int(self.reduce(0,op,l));
        }
    };
};
```

The following is an abstracted representation of a memory layout of the program generated by a hypothetical Cool compiler for the above code (note that this might or might not correspond to the layout generated by your compiler or the reference coolc):

Code segment:

$\texttt{maddr}_1$:	`cgen_m(BinOp.optype)`
$\texttt{maddr}_2$:	`cgen_m(BinOp.operate)`
$\texttt{maddr}_3$:	`cgen_m(SumOp.optype)`
$\texttt{maddr}_4$:	`cgen_m(MulOp.optype)`
$\texttt{maddr}_5$:	`cgen_m(MulOp.operate)`
$\texttt{maddr}_6$:	`cgen_m(IntList.tail_is_empty)`
$\texttt{maddr}_7$:	`cgen_m(IntList.get_head)`
$\texttt{maddr}_8$:	`cgen_m(IntList.set_head)`
$\texttt{maddr}_9$:	`cgen_m(IntList.get_tail)`
$\texttt{maddr}_{10}$:	`cgen_m(IntList.set_tail)`
$\texttt{maddr}_{11}$:	`cgen_m(IntList.generate)`
$\texttt{maddr}_{12}$:	`cgen_m(Main.reduce)`
$\texttt{maddr}_{13}$:	`cgen_m(Main.main)`

Dispatch tables:

$\texttt{maddr}_{14}$:	`DT[BinOp]`
$\texttt{maddr}_{15}$:	`DT[SumOp]`
$\texttt{maddr}_{16}$:	`DT[MulOp]`
$\texttt{maddr}_{17}$:	`DT[IntList]`
$\texttt{maddr}_{18}$:	`DT[Main]`

Stack ($\texttt{maddr}_{19}$) *(high address)*
↓

↑
Heap *(low address)*

In the above, $\texttt{maddr}_i$ represents the memory address at which the corresponding method's code or dispatch table starts. You should assume that the above layout is contiguous in memory. Note that the stack starts at a high address and grows towards lower addresses.

(a) The following is a representation of the dispatch table for class Main:

Method Idx	Method Name	Address
0	`reduce`	$\texttt{maddr}_{12}$
1	`main`	$\texttt{maddr}_{13}$

Provide equivalent representations for the dispatch tables of BinOp, SumOp, MulOp, and IntList. Assume that the Object class has no methods.

(b) Consider the state of the program at runtime when reaching (for the first time) the line marked with the comment "Point A". Give the object layout (as per Lecture 12) of every object currently on the heap which is of a class defined by the program (i.e., ignoring Cool base classes such as IO or Int).

For attributes, you can directly represent Int values by integers and an unassigned pointer by **void**. However, note that in a real Cool program, Int is an object and would have its own object layout, omitted here for simplicity. Finally, you can assume class tags are numbers from 1 to 5 given in the same order as the one in which classes appear in the layout above, and that attributes are laid out in the same order as the class definition.

(c) The following table represents an abstract view of the layout of the stack at runtime when reaching (for the first time) the line marked with the comment "Point A":

Address	Method	Contents	Description
$\texttt{maddr}_{19}$	`Main.main`	self	arg_0
$\texttt{maddr}_{19} - 4$	`Main.main`	...	Return
$\texttt{maddr}_{19} - 8$	`Main.main`	op	local
$\texttt{maddr}_{19} - 12$	`Main.main`	l	local
$\texttt{maddr}_{19} - 16$	`Main.main`	io	local
$\texttt{maddr}_{19} - 20$	`IntList.generate`	$\texttt{maddr}_{19} - 4$	FP
$\texttt{maddr}_{19} - 24$	`IntList.generate`	2	arg_1
$\texttt{maddr}_{19} - 28$	`IntList.generate`	self	arg_0
$\texttt{maddr}_{19} - 32$	`IntList.generate`	$\texttt{maddr}_{13} + \delta$	Return
$\texttt{maddr}_{19} - 36$	`IntList.generate`	l	local

Assume that the activation record is set as in Lecture 12 (e.g., slide 20). Note that we are assuming there are no stack frames above Main.main(...). This doesn't necessarily match a real implementation of the Cool runtime system, where main must return control to the OS or the Cool runtime on exit. For the purposes of this exercise, feel free to ignore this issue.

Since you don't have the generated code for every method above, you cannot directly calculate the return address to be stored on the stack. You should however give it as $\texttt{maddr}_i + \delta$, denoting an unknown address between $\texttt{maddr}_i$ and $\texttt{maddr}_{i+1}$. This notation is used in the example above. For locals, you should use the variable name, but remember that in practice it is the heap address that gets stored in memory for objects.

Give a similar view of the stack at runtime when reaching (for the first time) the line marked with the comment "Point B".

2. Consider the following arithmetic expression: $(1 + (2 * 3 - 8/4)) + (5 - 6) * 7$.

   (a) You are given MIPS code that evaluates this expression using a stack machine with a single accumulator register (similar to the method given in class Lecture 12). This code is wholly unoptimized and will execute the operations given in the expression above in their original order (e.g., it does not perform transformations such as arithmetic simplification or constant folding). How many times in total will this code push a value to or pop a value from the stack? Give a separate count for the number of pushes and the number of pops.

   (b) You are now given MIPS code that evaluates the same expression using a register machine with only 2 registers. Again, this code includes no optimizations and will perform every operation in the original expression, in the same order. How many loads from and stores to memory will this code perform, at a minimum? Give a separate count for loads and for stores.

3. Suppose you want to add a for-loop construct to Cool, having the following syntax:

$$\text{for } e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof}$$

The above for-loop expression is evaluated as follows: e_1 is evaluated first, e_2 is evaluated next, and then the body of the loop (e_3) is executed once for every integer in the range $[e_1, e_2]$ (inclusive) in increasing order. Here e_1 and e_2 should be evaluated only once. Similar to the while loop, the for-loop returns void.

(a) Give the operational semantics for the for-loop construct above.

(b) Give the code generation function cgen(for e_1 to e_2 do e_3 rof) for this construct. Use the code generation conventions from the lecture. The result of cgen(...) must be MIPS code following the stack-machine with one accumulator model.

Assume that cgen(e_1) (and similarly cgen(e_2)) does integer unboxing, i.e., the evaluation result of e_1 will be stored in `$a0` after executing cgen(e_1). You can use the instruction `ble r1, r2, label` in order to branch to `label` if `r1` $\leq$ `r2`.

4. Consider the following basic block, in which all variables are integers.

```
        a := f / 1
        b := f * g
        c := b + b
        d := a - 3
        x := a * g
        y := x + x
        z := y * c
```

(a) Assume that the only variable that is live at the exit of this block is `z`, while `f` and `g` are given as inputs. In order, apply the following optimizations to this basic block. Show the result of each transformation. For each optimization, you must continue to apply it until no further applications of that transformation are possible, before writing out the result and moving on to the next. Assume that `+` is faster than `*` and `<<`.

   i. Algebraic simplification
   ii. Copy propagation
   iii. Common sub-expression elimination
   iv. Copy propagation
   v. Dead code elimination

(b) The resulting program is still not optimal. What optimizations, in what order, can you apply to fully optimize the result? Show the maximally optimized codes (with least number of instructions).

5. Consider the following assembly-like pseudo-code, using 11 temporaries (abstract registers) `t0` to `t10`:

```
t1 := t0 * t0
t2 := -t1
t3 := t2 + t0
if t3 > 0:
    t4 := t0 - t3
    t5 := t2 * 3
    t6 := t4 - t5
else:
    t7 := t0 + t3
    t8 := t2 * 3
    t6 := t7 + t8
t9 := t0 * t3
t10 := t6 + t9
```

   (a) At each program point, list the variables that are live. Note that `t0` is the only input for the given code and `t10` is the only live value on exit.

   (b) Draw the register interference graph between temporaries in the above program as described in class.

   (c) Provide a lower bound on the number of registers required by the program that does not involve spilling, based on (a) or (b). Can you explain why?

   (d) Using the algorithm described in class, provide a coloring of the graph in (b). The number of colors used should be your lower bound in (c). Provide the final k-colored graph (you may use the tikz package to typeset it or simply embed an image), along with the order in which the algorithm colors the nodes.

   (e) Based on your coloring, write down a mapping from temporaries to registers (labeled `r0`, `r1`, etc.).