Assignment 8: Green Diamondback

Rohan Puthukudy

June 11, 2023

1 Approach to Proper Tail Calls

My appraoch to proper tail call optimization involved first flagging moments of the snek code that contain tall calls during the compilation step. This consisted of a `bool` variable passed in the `compile_expr` function.

Then, the next part of my approach is to load new arguments into existing locations on the stack and performing jumps instead of calls when making recursive *tail* calls. This is seen in the following code.

```
if tail {
    // Tail Calling Convention
    // 1. Push each newly computed arg onto the stack
    // 2. Pop each of those values into the right spots (in reverse order)
    // 3. Jump back to the body of our function

    for (i, arg) in args.iter().enumerate() {
        self.compile_expr(cx, Loc::Reg(Reg::Rax), arg, false);
        self.emit_instr(Instr::PushR(Reg::Rax));
    }

    for i in 0..args.len() {
        self.emit_instr(Instr::Pop(Loc::Mem(mref![Rbp + %(8 *
            (args.len()-i+1))])));
    }

    self.emit_instr(Instr::Jmp(fun_body_label(*fun)));
}
```

Here, we see that my proper tail call convention consists of the following steps, taken after determining that we are indeed in tail call position.

1. Push each newly computed argument value onto the stack
2. Pop all of these values off the stack and into the existing locations for arguments in our stack frame. This is done in reverse order.

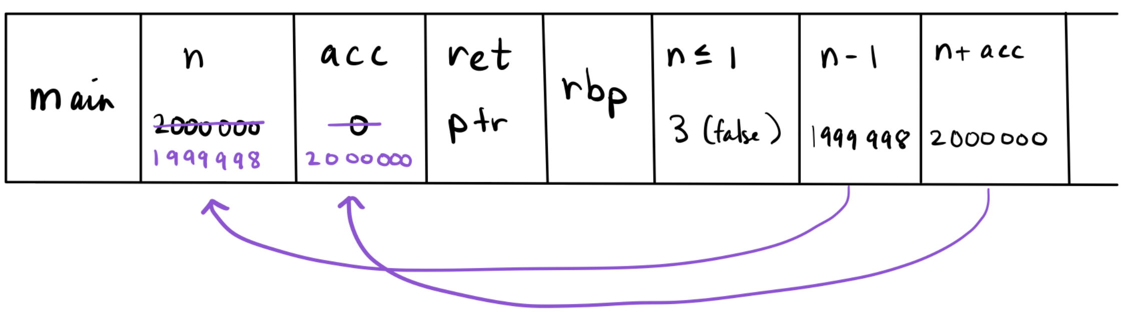
3. Finally, issue a *jump* back to the body of our function, instead of performing a recursive call

2 Memory Diagram

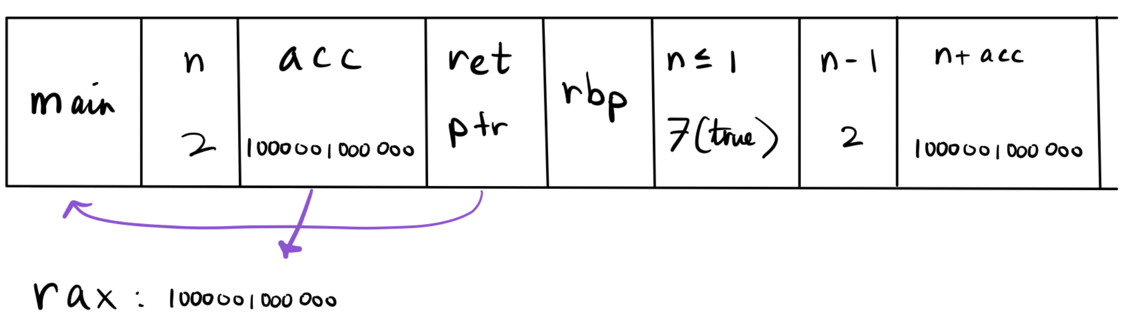
Here is a diagram depicting the stack before, during, and when returning from the first tail call made in `tail1.snek`.

Before Tail Call

During the Call



Returning From the Call



3 Tail Call Tests

3.1 Test 1

Our first test is the following

```
(fun (sum n acc)
    (if (< n 1)
        acc
        (sum (sub1 n) (+ n acc))
    )
)

(sum 1000000 0)
```

This program recursively computes the sum of the first n (or 1000000 in our case) numbers, using an accumulator and recursive tail calls.

When compiled and run, it outputs `500000500000` as desired.

3.2 Test 2

Our second test is the following

```
(fun (even n)
    (if (= n 0)
        true
        (odd (sub1 n))
    )
)

(fun (odd n)
    (if (= n 1)
        true
        (even (sub1 n))
    )
)

(even 1000000)
```

This program determines if the inputted integer (1000000 in our case) is even or not. The even function uses the odd function as a helper function and they call each other mutual recursively to arrive at the correct answer.

When compiled and run, this program outputs `even` as desired.

3.3 Test 3

Our third test is the following

```
(fun (sum-of-squares acc n)
    (if (= n 1)
        (+ acc 1)
        (sum-of-squares (+ acc (* n n)) (sub1 n))
    )
)
(sum-of-squares 0 1000000)
```

This program recursively computes the sum of the squares of the first n (or 1000000 in our case) positive integers, using an accumulator and recursive tail calls. In other words, it computes $\sum_{k=1}^{n} k^2$ for any inputted $n \geq 1$.

When compiled and run, it outputs `333333833333500000` as desired.

4 Resources Used

I used multiple different websites as references for calling conventions, tail calls, and x86_86 assembly. I've listed them below.

- https://course.ccs.neu.edu/cs4410sp20/lec_tail-calls_stack_notes.html
- https://eklitzke.org/how-tail-call-optimization-works
- https://courses.cs.cornell.edu/cs3110/2021sp/textbook/data/tail_recursion.html
- https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf