1 Introduction to Tail Recursion

Let's consider the following function:

```
(fun (sumrec num sofar)
    (if (= num 0)
        sofar
        (sumrec (+ num -1) (+ sofar num))
    )
)
```

This function is in tail position: that is, after the recursive call, we don't need to do any additional computations. The assembly representation, is shown to the left. On the right, the stack frame when `(subrec 3 0)` is evaluated is shown, at the point when the base case is about to be executed.

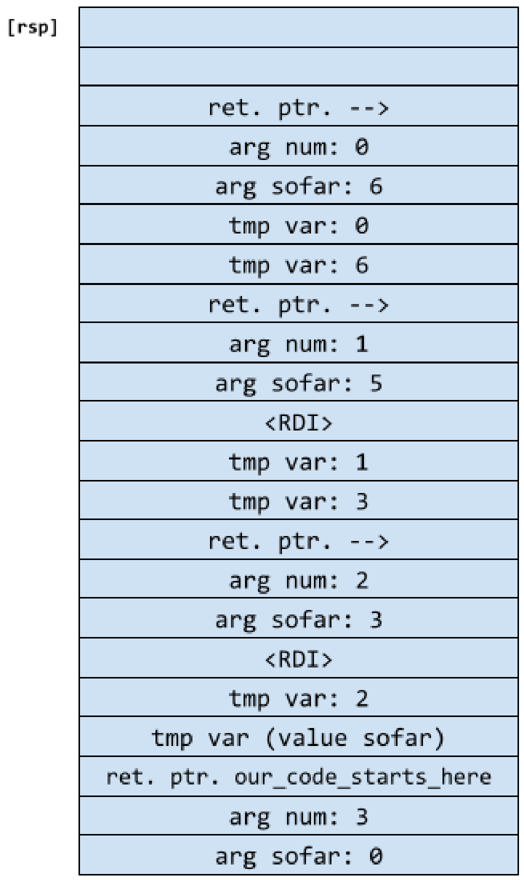
```
sumrec:
    sub rsp, 16

    mov rax, [rsp + 24]
    mov [rsp + 0], rax
    ... if (= num 0)
    cmp rax, 1
    je ifelse_1
        mov rax, [rsp + 32]
        jmp ifend_0
    ifelse_1:
        mov rax, [rsp + 24]
        ... add -1 to num,
        ... store on stack as tmp
        mov [rsp + 0], rax

        mov rax, [rsp + 32]
        ... add sofar to num,
        ... store in rax
        add rax, [rsp + 8]

        ... 2-arg calling
        ... convention from class
        sub rsp, 24
        mov rbx, [rsp+24]
        mov [rsp], rbx
        mov [rsp+8], rax
        mov [rsp+16], rdi
        call sumrec
    --> mov rdi, [rsp+16]
        add rsp, 24

    ifend_0:
    add rsp, 16
    ret
```



(Exercise.) What will `RAX` be after each return pointer?

> `RAX` will be **6**. There are two ways you can tell:
>
> - In the code itself, the base case just returns the `sofar` argument. Intuitively, this means that this function should return 6 for all subcalls, implying `rax` has 6 at every return pointer.
> - Additionally, when looking at the generated assembly, notice the `move rax, [rsp + 32]` instruction. `[rsp + 32]` points to the value `sofar`, 6. Pair this with when we look at the instructions after the `call` instruction (which is where the program returns to after `ret` is executed), notice that there's no additional move instructions to `rax`.

Remark: There are several things to notice:

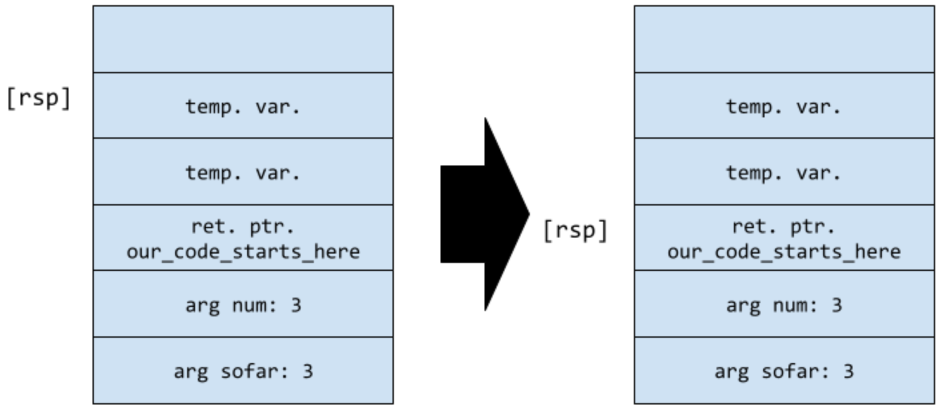
- there's no use of local variables or arguments following the `call` instruction.
- there are no changes to `rax` after the `call` instruction.

These observations means that we can reuse the space that we set up for the function call to perform all operations. In other words, we can use one stack frame's worth of space for the *entire* computation! Basically, we're using more space than we need.

1.1 Restructuring the Assembly

Instead of the `call` instruction above, what should we do to replace the instructions so that the new instructions ovewrite the current arguments with the new arguments and "re-use" the stack frame?

We can preemptively add to `rsp` to "undo" the `sub` at the beginning of the function. That is,

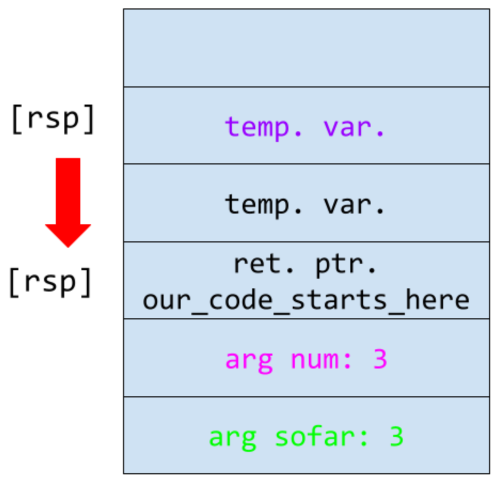


So, we can write the following assembly:

```
add rsp, 16
mov rbx, [rsp - 16]
mov [rsp + 8], rbx
mov [rsp + 16], rax
jmp sumrec
```

Notice how we have an *unconditional jump* to `sumrec`, as opposed to a `call` instruction. This is important, because this effectively means we have a **loop**!

```
add rsp, 16
mov rbx, [rsp - 16]
mov [rsp + 8], rbx
mov [rsp + 16], rax
jmp sumrec
```



This process is known as **proper tail calls**. We effectively re-use the stack from when the “last thing” is a function call (what would have been a separate function call with its own stack frame is now a separate “function call” re-using the same stack frame from the current “function call.”).

1.2 Tail Call Positions

When is an expression in tail call position? How does the compiler know when it is in tail call position? A few things:

- Anything where we generate instructions that work with the result of a subexpression is not in tail call.
- Essentially, if you work with `rax` or store the value after a recursive call to the compile function, that recursive function call cannot be in tail call position. If you don’t do anything with it and rely on the answer being in `rax` after making the recursive call, then it’s in tail position.

So, with this in mind:

- `Add1`: not in tail call position (adds 1 to `rax`).
- `Eq`: not in tail call position (needs to store `rax` somewhere before compiling the second expression so we can compare them).
- `Plus`: not in tail call position (needs to store `rax` somewhere so we can add to `rax` later).
- `Let`: the value associated with the binding cannot be in tail position, but the body of the `let`-expression *can* be in tail position (provided that any preceding calls are in tail position as well).
- `If`: the conditional expression cannot be in tail position, but the then/else expressions can be in tail position (provided that any preceding calls are in tail position as well).
- `Block`: only the last expression in a block can be in tail position; the other expressions cannot be in tail position.
- `Break`/`Loop`: they *generally* cannot be in tail call position (there are special cases, though).

In a function call, the arguments cannot be in tail-call position. Note that the body of the expression is where we compile tail call position code or regular code based on whether we’re in tail call position.

In other words, at the beginning of compilation, we assume that we’re in tail call position. As we compile each expression, we might “lose” that tail call position. We can never get it back once lost in that subexpression.