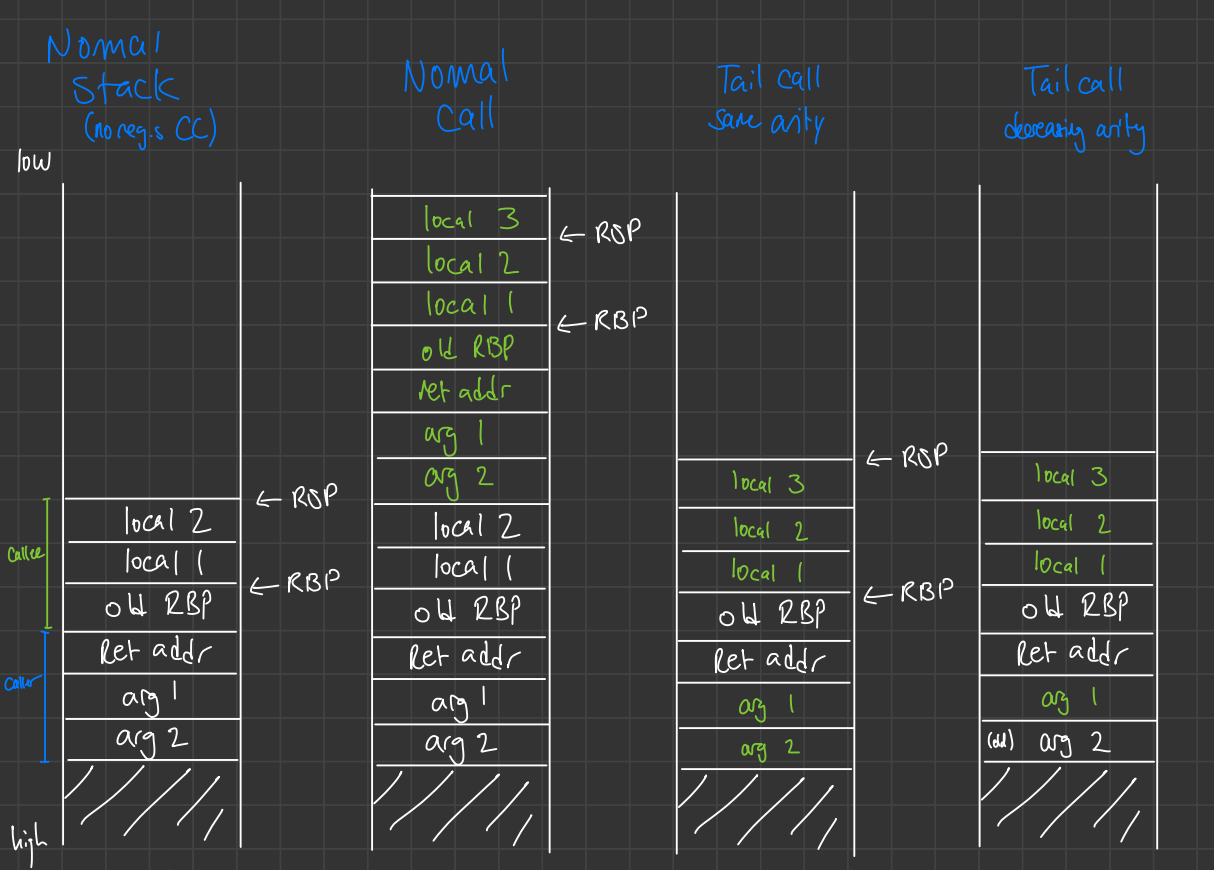


STARTING AT 1:

local i at RBP - 8*i

-8, -16, -24

args at RBP + 8(i+1)

+16, +24, +32

GOAL: slot in args in the old arg slots, keeping save ret addr & saved RBP

Idea: just slot in new args & ignore the extra reserved space

Note: can use more locals, so reset RSP to RBP before jumping and first thing → reserve space

- we get to scribble over old locals

* or some cycle detection thing... prob not worth

Implementation note: if any of the values we are slotting in as new args are themselves coming from arguments (specifically, ones we are overwriting), we have to push → pop into place

* TODO: how to check this safety condition?

- look @ arity → determine which args are being over-written → before pulling out, check if one of those args?

Compiling a function decl (basic)

```
START:
    push RBP
    mov RBP, RSP
    sub RSP, 8*N      ← (parity-adjusted) deepest stack for locals,
    :
END:
    mov RSP, RBP
    pop RBP
    ret
```

Compiling a function call (basic)

```
    push arg_M
    :
    push arg_2
    push arg_1
    call func
    add RSP, 8*M      ← clear off args that got pushed
```

Compiling our function decls

* TODO: how to make this fit in w/ our current helpers

```
foo:                  ← where we jump when we 'call'
                        (remember to add to RSP after calls return)
foo_prologue:
    push RBP
    mov RBP, RSP
foo_body:             ← the first thing the body now does
    sub RSP, 8*N        (aka where we jump to for tail call)
    :                   is reserve space on the stack
```

Returning (base case)

```
    mov RSP, RBP
    pop RBP
    ret
```

jumping (tail call)

```
    mov RSP, RBP      ← restore used stack space before making tail call, see above
    jmp bar_body
```

Creates w/ lambdas bc we don't know what they call at compile time effectively

Normal Stack (no reg.s CC)

Scenario:
- F (normal) calls G, pushing on N args
- G (tail) calls H, w/ greater arity (M)
  - if we shifted everything up here, when we return, F only adds N to RSP, so (M-N) args are still there

Idea 1: pad all function calls to max called arity
- G would have to be called w/ M arg slots, the bottom (M-N) of which are padding until H slots things in there
- pragmatics: lots of stack space usage (always max possible, along any path of calls)
- implementation: would basically have to construct a directed graph of function calls
  - find all SCCs → max w/i SCC → build DAG of SCCs
  - could be hard to compute w/o mutation (should be O(V+E))
  - where to store decorated data?
  - does this break for lambdas? ⇒ YES!

Idea 2: Change the calling convention

From write-up: save the RSP?
- we need to know saved RSP, RBP, and return address
  - everything else can be calculated from those
- both parameters and locals will be above RBP

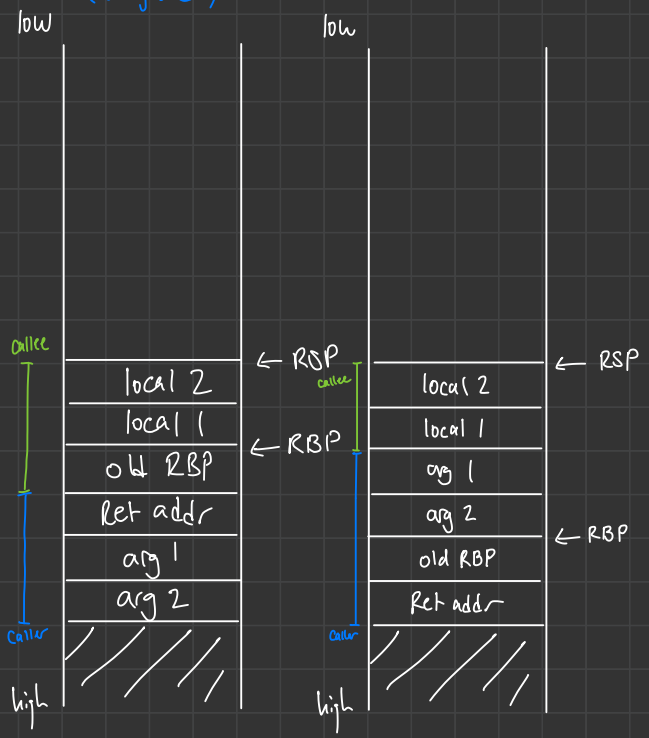
We won't be able to use call

Maybe we could use 3 calling conventions:
- one for external calls
- one for internal calls
- one for internal tail calls

Probably just use the same convention for all internal

Problem: where do we find args? What if we have a function being called in both tail & non-tail

Normal Stack (no reg.s CC)



Callee responsibility:
- load in locals
- stash the old RBP — not necessarily?
- return to ret address
- callee-save reg.s:
  - RSP, RBP

Caller responsibility:
- load in args
- load in ret address
- caller-save registers
- reclaim stack space by RSP +=

1. Load new args onto stack
- these may use args or locals, and may overwrite args (new) or locals → Safety analysis like above ✓

What does this fix?

- now, when the callee returns, the entire stack (locals AND args) get paired down ← (no longer have to worry about missing the bottom M-N args)

Compiling a function call (basic)

Push RBP
load RSP→RBP

```
push return address   ← probably hard to do [JUST ADD LABEL!]
push arg_M
:
push arg_2
push arg_1
jump foo
```

Stack looks like

JUMPS

```
sub RSP, 8*N
:
(body)
END
mov RSP, RBP
pop RBP
ret
```

Problem:
- when we want to use more locals, we reset RSP below locals & above args and re-decrement

ANS: just reset to RBP + arity of next function

Still have to figure out how to do the damn tail call...