CSE 231: Tail Calls

Ranjit Jhala *April 30, 2026*

Recursion is Awesome!

Lets us **decompose** problems into smaller *sub-problems*!

QUIZ: What happens when we run?

What happens if we compile and run these (silly) functions?

```
(fun (foo n)
  (if (<= n 0)
    99
    (foo (sub1 n))
  )
)

(foo input)
```

```
$ make foo.run

$ ./foo.run 100
_____________________

$ ./foo.run 1000000
_____________________
```

```
(fun (sum n acc)
  (if (= n 0)
    0
    (+ n (sum (sub1 n)))
  )
)

(sum input)
```

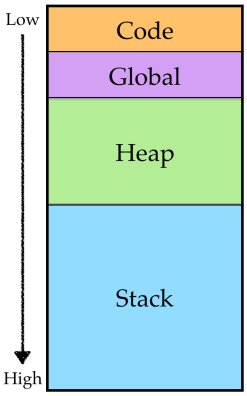
```
$ make sum.run

$ ./sum.run 100
_____________________

$ ./sum.run 1000000
_____________________
```

Visualizing Calls

Lets **visualize** the calls to `sum` (on a smaller input!)

```
    sum 5
==> <(+ 5 <(sum 4)>)>
==> <(+ 5 <(+ 4 <(sum 3))>)>)>
==> <(+ 5 <(+ 4 <(+ 3 <(sum 2)>)>)>)>
==> <(+ 5 <(+ 4 <(+ 3 <(2 + <(sum 1)>)>)>)>)>
==> <(+ 5 <(+ 4 <(+ 3 <(2 + <(+ 1 <(sum 0)>)>)>)>)>)>
==> <(+ 5 <(+ 4 <(+ 3 <(2 + <(+ 1 <0>)>)>)>)>)>
==> <(+ 5 <(+ 4 <(+ 3 <(2 + <1>)>)>)>)>
==> <(+ 5 <(+ 4 <(+ 3 <3>)>)>)>
==> <(+ 5 <(+ 4 <6>)>)>
==> <(+ 5 <10>)>
==> 15
```

The <. . .> indicate the "stack frames"; how deep does the stack grow?



QUIZ: How many stack frames?

Created when we run `(sum 1000000)`? _____

Problem with Recursion

- `sum n` has to **wait** for the result of `sum (sub1 n)`
- can only add `n` **after** the result comes back!

Recursion with Accumulation

Lets rewrite `sum` to **accumulate** the result as we go ...

```
(fun (helper n acc)
  (if (= n 0)
    acc                          ; just return `acc`
    (helper (sub1 n) (+ acc n)) ; accumulate result
  ))

(fun (sum n)
  (helper n 0))
```

The `acc` parameter **accumulates** the result as we recurse.

Visualizing Accumulating Calls

Now, what happens when we run `(sum 5)`?

Vanilla Recusion

```
    sum 5
==> <helper 5 0>
==> <<helper 4 5>>
==> <<<helper 3 9>>>
==> <<<<helper 2 12>>>>
==> <<<<<helper 1 14>>>>>
==> <<<<<<helper 0 15>>>>>>
==> <<<<<<15>>>>>>
==> <<<<<15>>>>>
==> <<<<15>>>>
==> <<<15>>>
==> <<15>>
==> <15>
==> 15
```

Optimized

```
    sum 5
==> <helper 5 0>
==> <helper 4 5>
==> <helper 3 9>
==> <helper 2 12>
==> <helper 1 14>
==> <helper 0 15>
==> 15
```

- No need to **wait** for result of recursive call
- It just gets **returned** as our result!

Idea: Reuse Stack Frames!

As we don't have to wait for the result of `helper (sub1 n)` we can just **reuse** the same stack frame for the recursive call!

QUIZ: Modify Assembly to Reuse Stack Frame

Source

```
(fun (helper n acc)
  (if (= n 0)
    acc
    (helper (sub1 n) (+ acc n))))
```

Original Assembly

```
fun_start_helper:
push rbp                ; SETUP FRAME
mov rbp, rsp
sub rsp, 8*3

mov rax, [rbp - 8*-2]   ; IF (= n 0)
mov [rbp - 8*1], rax
mov rax, 0
cmp rax, [rbp - 8*1]
mov rcx, 3
mov rdx, 1
cmove  rax, rcx
cmovne rax, rdx
cmp rax, 1
je if_false_0           ; THEN BRANCH
if_true_0:
mov rax, [rbp - 8*-3]
jmp if_exit_0
if_false_0:             ; ELSE BRANCH
mov rax, [rbp - 8*-2]
sub rax, 2
mov [rbp - 8*1], rax    ; (sub1 n)
mov rax, [rbp - 8*-3]
mov [rbp - 8*2], rax
mov rax, [rbp - 8*-2]
add rax, [rbp - 8*2]    ; (+ acc n)

push rax                ; (helper ..)
mov rax, [rbp - 8*1]
push rax
call fun_start_helper
add rsp, 16
if_exit_0:

mov rsp, rbp            ; TEARDOWN
FRAME
pop rbp
ret
```

Optimized Assembly

```
?

?
```

Strategy

Step 1: Identify "tail" calls that can reuse the stack frame
Step 2: Modify assembly to use `jmp` instead of `call`!

QUIZ: Tail Calls

A **tail call** is a function call that is the **last** thing executed in a function, i.e. there is **no more work** after the call returns.

Which of the two calls below are *tail calls*?

```
(fun (fac n acc)
  (if (= n 0)
    acc
    (if (= n 2)
        (* 2 (fac (sub1 n) (* acc n)))
        (fac (sub1 n) (* acc n))
    )
  )
)
```

QUIZ: Which Expressions Can Have Tail Calls?

- **Circle** the expression that *can have* tail calls
- **Cross out** the expressions that *cannot have* tail calls

```
e ::= n
    | true
    | false
    | input
    | x
    | (add1 e)
    | (let (x e1) e2)
    | (+ e1 e2)
    | (= e1 e2)
    | (if e1 e2 e3)
    | (set x e)
    | (block e1...en)
    | (loop e)
    | (break e)
    | (print e)
    | (call1 f e)
    | (call2 f e1 e2)
```

Step 1: Identify Tail Calls in Compilation

```
fn compile(e: &Expr, env: &Stack, lbl:&str, tr: bool) -> String {
  match e {
    Add1(e1) => {
      let e1_code = compile(e1, env, lbl,             );
      ...
    }
    Plus(e1, e2) => {
      let e1_code = compile(e1, env, lbl,         );
      let e2_code = compile(e2, env, lbl,         );
      ...
    }
    Eq(e1, e2) => {
      let e1_code = compile(e1, env, lbl,         );
      let e2_code = compile(e2, env, lbl,         );
      ...
    }
    Let(x, e1, e2) => {
      let e1_code = compile(e1, env, lbl,         );
      let e2_code = compile(e2, &newenv, lbl,     );
      ...
    }
    If(cond, e_then, e_else) => {
      let cnd_code = compile(e_cnd, env, lbl,        );
      let thn_code = compile(e_then, env, lbl,        );
      let els_code = compile(e_else, env, lbl,        );
      ...
    }
    Set(x, e) => {
      let e_code = compile(e, env, lbl,        );
      ...
    }
    Block(es) => {
      let n = es.len();
      let e_codes: Vec<String> = es.iter().enumerate()
          .map(|(i, e)| compile(e, env, lbl,         ))
          .collect();
      ...
    }
    Loop(e) => {
      let e_code = compile(e, env, &loop_exit, , f);
      ...
    }
    Break(e) => {
      let e_code = compile(e, env, lbl, , f);
      ...
    }
    Print(e) => {
      let e_code = compile(e, env, lbl, , f);
      ...
    }
    Call2(f, e1, e2) => {
      let e1_code = compile(e1, env, lbl, , f);
      let e2_code = compile(e2, env, lbl, , f);
      ...
    }
  }
}
```

Step 2: Modify Assembly to Use `jmp` for Tail Calls

Next, lets **modify** the assembly to use `jmp` instead of `call`!

Definitions

```
fn compile_defn(&mut self, defn: &expr::Defn) -> String {
  let label = entry_label(&defn.name);

  let entry = compile_entry(&defn.body);
  let stack = Stack::new(&defn.params[..]);
  let exit = exit_label();
  let body = self.compile_expr(&defn.body, &stack, &exit, true);
  let exit = compile_exit();

  format!(
    "{label}:\n\
     {entry}\n\

     {body}\n\
     {exit}"
  )
}
```

Calls

```
fn compile(&mut self, e: &Expr, env: &Stack, lbl: &str, tr: bool)
  -> String
{
  match e {
    ...
    Call2(f, e1, e2) => {
      let e1_code = self.compile(e1, env, lbl, false);
      let e2_code = self.compile(e2, env, lbl, false);
      let call_code = if tr {
        format!(
          "{e1_code}\n\

           {e2_code}\n\



          "
        )
      } else {
        ...
      };
    }
  }
}
```

Pitfalls to Avoid

You have to implement tail calls for *mutually recursive* functions with *multiple* arguments. What problems do you foresee?