CSE 331 Software Design & Implementation

Winter 2025 Section 7 – Tail Recursion

Administrivia

• HW 7 written released tonight, due Wed. Feb 26

Loops vs Tail Recursion

• Tail-call optimization turns tail recursion into a loop

Loops **Solution** State **Sector** Stat

- •Tail recursion can solve all problems loop can
- -any loop can be translated to tail recursion
- -both use O(1) memory with tail-call optimization
- •Translation is simple and important to understand
- •Tells us that Loops << Recursion –correspond to the *special* case of tail recursion

Loop to Tail Recursion

Translate loop to tail recursive helper function and main function:

```
const myLoop = (R: List): T => {
    let s = f(R);
    while (R.kind !== "nil") {
        s = g(s, R.hd);
        R = R.tl;
    }
    return h(s);
    my-acc(nil, s) := h(s)
    my-acc(nil, s) := h(s)
```

- };
- **1.** Loop body \rightarrow recursive case of accumulator function
- **2.** After loop body \rightarrow base case of accumulator function
- **3.** Before loop body \rightarrow variable set up

Loop to Tail Recursion

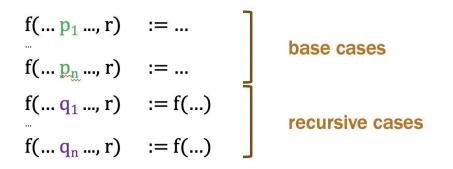
```
const myLoop = (R: List): T => {
  let s = f(R);
  while (R.kind !== "nil") {
    s = g(s, R.hd);
    R = R.tl;
  }
  return h(s);
};
```

 Final result: tail-recursive function that does same calculation: my-func(R) := my-acc(R, f(R))
 Main func to call

```
my-acc(nil, s) := h(s)my-acc(x :: L, s) := my-acc(L, g(s, x))
```

Helper accumulator func

Tail Recursion to Loop



Tail-recursive function becomes a loop:

```
// Inv: f(args<sub>0</sub>) = f(args)
while (args /* match some q pattern */) {
    args = /* right-side of appropriate q pattern */;
}
return /* right-side of appropriate p pattern */;
```

Rewriting the Invariant

```
// Inv: sum-acc(S<sub>0</sub>, r<sub>0</sub>) = sum-acc(S, r)
while (S.kind !== "nil") {
    r = S.hd + r;
    S = S.tl;
}
return r;
```

- This is the most direct invariant
 - says answer with current arguments is the original answer
- Can be rewritten to not mention sum-acc at all
 - use the relationship we proved between sum-acc and sum

Digit representations: List<Z>

Example, 120 in Base-10:

"Big endian": 1 :: 2 :: 0 :: nil

- higher order digits at the front

"Little endian": 0 :: 2 :: 1 :: nil 👞

- higher order digits at the end

We're using this one

Defining value of a base-b digit as: value(nil, b) := 0 value(d :: ds, b) := $d + b \cdot value(ds)$

 $\begin{aligned} \mathsf{value-acc}(\mathsf{nil},\,b,\,c,\,s) & := s \\ \mathsf{value-acc}(d::\mathsf{ds},\,b,\,c,\,s) & := \mathsf{value-acc}(\mathsf{ds},\,b,\,b\cdot c,\,s+c\cdot d) \end{aligned}$

Write a function that calculates value-acc(digits, b, 1, 0) with a **loop**. Your function should have the following signature:

const value = (digits: List<number>, base: number): number => { ... };

Be sure to include the invariant of the loop!

Write a function that calculates value-acc(digits, b, 1, 0) with aloop.value-acc(nil, b, c, s):= s

 $\mathsf{value}\mathsf{-acc}(d :: \mathsf{ds}, \, b, \, c, \, s) \;\; := \; \mathsf{value}\mathsf{-acc}(\mathsf{ds}, \, b, \, b \cdot c, \, s + c \cdot d)$

Prove that value-acc(ds, b, c, s) = s + c * value(ds, b)

 $\mathsf{value-acc}(\mathsf{nil},\,b,\,c,\,s) \qquad := \ s$

 $\mathsf{value}\mathsf{-}\mathsf{acc}(d::\mathsf{ds},\,b,\,c,\,s) \ := \ \mathsf{value}\mathsf{-}\mathsf{acc}(\mathsf{ds},\,b,\,b\cdot c,\,s+c\cdot d)$

value(nil, b) := 0 value(d :: ds, b) := $d + b \cdot value(ds)$

Prove that value-acc(ds, b, c, s) = s + c * value(ds, b)

value-acc(nil, b, c, s) := s

 $\mathsf{value}\mathsf{-acc}(d::\mathsf{ds},\,b,\,c,\,s) \;\; := \;\; \mathsf{value}\mathsf{-acc}(\mathsf{ds},\,b,\,b\cdot c,\,s+c\cdot d)$

value(nil, b) := 0 value(d :: ds, b) := $d + b \cdot value(ds)$

Use equation value-acc(ds, b, c, s) = s + c * value(ds, b)

to rewrite the invariant so that it no longer mentions "value-acc".

// Inv: value-acc(digits_0, base, 1, 0) = value-acc(digits, base, c, s)

Question 4a

Invariant: value(digits_0, base) = s + c * value(digits, base)

Prove that the invariant holds at the top of the loop

Question 4b

Invariant: value(digits_0, base) = s + c * value(digits, base)

Prove that, when we exit, the function returns value(digits_0, base)

Question 4c

Invariant: value(digits_0, base) = s + c * value(digits, base) Prove that the invariant holds when we first get to the top of the loop.