CSE 331
Software Design & Implementation

Autumn 2024

Section 7 – Tail Recursion

Administrivia

- HW 7 written released tonight, due Friday 11/15 at 11pm (it’s shorter than HW6 you guys will be ok!)
- HW 7 coding released Thursday, due Monday 11/18 at 11pm

Loops vs Tail Recursion

- Tail-call optimization turns tail recursion into a loop

Loops ≤ Tail Recursion (with tail-call optimization)

•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-func(L) := my-acc(L, f(L))

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

my-acc(nil, s) := h(s)

1. Loop body → recursive case of accumulator function
2. After loop body → base case of accumulator function
3. Before loop body → 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(L) := my-acc(L, f(L)) Main func to call

  my-acc(nil, s) := h(s)

  Helper accumulator func

Tail Recursion to Loop

f(… p_1 …, r)	:= …	base cases
…		
f(… p_n …, r)	:= …	
f(… q_1 …, r)	:= f(…)	recursive cases
…		
f(… q_n …, r)	:= f(…)	

- **Tail-recursive function becomes a loop:**

```
// Inv: f(args0) = 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(S0, r0) = 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

Question 1

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

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

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

Be sure to include the invariant of the loop!

Question 1

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

```
const valueAcc = (digits: List<number>, base: number): number => {


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







};
```

Question 2

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

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

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

$$\text{value}(\text{nil}, b) \quad := \quad 0$$

$$\text{value}(d :: \text{ds}, b) \quad := \quad d + b \cdot \text{value}(\text{ds})$$

Question 2

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

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

$$\begin{aligned}\mathsf{value}(\mathsf{nil}, b) &:= 0 \\ \mathsf{value}(d :: \mathsf{ds}, b) &:= d + b \cdot \mathsf{value}(\mathsf{ds})\end{aligned}$$

Question 3

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”.

Question 4a

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.

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 is preserved when we execute the loop body.