Tail recursion and invariants

Ziyan Maraikar

July 10, 2014

Table of Contents

Memory use during recursion

```
fact 4
≡ 4 * fact 3
≡ 4 * 3 * fact 2
≡ 4 * 3 * 2 * fact 1
≡ 4 * 3 * 2 * 1
```

- ★ Observe how the list of intermediate values grows longer as the computation progresses.
- ★ Each intermediate value must be retained until the recursion terminates.
- ★ This corresponds to the stack space used during the recursive evaluation.

Forms of recursion

Do you notice anything different about the way that evaluation of functions **gcd** and **fact** occur?

```
fact 4
≡ 4 ∗ fact 3
≡ 4 ∗ 3 ∗ fact 2
≡ 4 ∗ 3 ∗ 2 ∗ fact 1
≡ 4 ∗ 3 ∗ 2 ∗ 1
```

```
gcd 12 33
≡ gcd 33 9
≡ gcd 9 6
≡ gcd 6 3
≡ gcd 3 0
≡ 3
```

During the evaluation of **gcd** the size of expression remains the same, that is, it requires a fixed amount of space.

FIBONNACCI

$$
\mathit{fib}\ n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathit{fib}(n-1) + \mathit{fib}(n-2) & \text{otherwise} \end{cases}
$$

```
let rec fib n =
  if n=0 then 0
  else if n=1 then 1
  else fib (n-1) + fib (n-2)
```

Redundant computations in fib

```
fib 3
≡ if 3=0 then ... else fib 2 + fib 1
≡ fib 2 + fib 1
≡ (if 2=0 then ... else fib 1 + fib 0) + fib 1
≡ fib 1 + fib 0 + fib 1
≡ (if 1=0 then 0 else if 1=1 then 1 else ...) + fib 0 + fib
    1
≡ 1 + fib 0 + fib 1
≡ 1 + (if 0=0 then 0 else ...) + fib 1
≡ 1 + 0 + fib 1
≡ 1 + 1 + (if 1=0 then 0 else if 1=1 then 1 else ...)
≡ 1 + 0 + 1
```

Observe how many times we compute `fib 1`. This repetition increases as `n` grows.

Code shape

- Some recursive definitions grow larger during each step of the evaluation, e.g. factorial, Fibonnacci.
- Others like Euclid's GCD stay constant. Can we convert all functions to this form?
- Some recursive definitions needlessly recompute results, e.g. Fibonnaci, Taylor expansion. Can we avoid this?

Tail recursion

Definition

A recursive function is tail recursive if the final result of the recursive call is the final result of the function itself.

- ★ If the result of the recursive call must be further processed (say, by adding 1 to it), it is not tail recursive.
- ★ At a tail call, the containing function is about to return, so we don't need to save any intermediate values.
  Therefore, we can perform *tail call elimination* — the recursive call can be entered without creating a new stack frame.

Conversion to tail recursive form

- During evalutaion of a non-tail recursive function we have to keep a chain of intermediate results until the recursive application provides the result.
- Instead we can carry this intermediate result forward using an additional function parameter (called an *accumulator*.)
- At each step we update the value of the accumulator so that at termintation, the final result is in the accumulator.

Example

```
let rec tr_fact
  (n:int) (a:int) :int =
  if n=0 then a
  else tr_fact (n-1) (n*a)
```

```
int tr_fact (int n) {
  int a=1;
  while (n>1) {
    a *= n;
    n--
  }
  return a;
}
```

Note how the loop variable `a` in the C version corresponds to the accumulator paramter that we introduced.

Exercise

- Show the evaluation of `tr_fact 5 1`.
- State the relationship between the initial argument n_0, n and a.

Hiding implementation details

The accumulator in our tail-recursive factorial is not relavant to its clients. We can hide it by nesting **tr_fact** within a factorial function.

```
let fact (n0:int) :int =
  let rec tr_fact (n:int) (a:int) :int =
    if n=0 then a
    else tr_fact (n-1) (n*a) in
  tr_fact n0 1
```

Table of Contents

Tail recursive coding style

```
let fact (n:int) :int =
(* Compute factorial using tail recursion *)
  let rec loop (i:int) (a:int) :int =
    if i=0 then a
    else loop (i-1) (a*i)
  in loop n 1
```

- ★ Use a comment to describe *what* you code is doing.
- ★ The tail-recursive helper function is usually called **`loop`**.
- ★ Observe that **`loop`** can access the parameter **`n`** in the outer function.

Reasoning about tail recursion

$$n! = \begin{cases} 1 & n = 1 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

```
let fact (n:int) :int =
let rec loop
  (i:int) (a:int) :int =
  if i=0 then a
  else loop (i-1) (a*i)
in loop n 1
```

How do we know that the tail-recursive version actually calculates $n!$?

Mathematical induction

1. Devise a formula relating the loop index i, the accumulator a, and the function's result. This is called the *invariant*.
2. Show that the invariant holds true at the initial iteration.
3. Assume that the invariant holds at an arbitrary iteration i, and show that it remains valid at the next iteration.
4. Show that the accumulator a contains the desired value when tail recursion terminates.

Example

```
let fact (n:int) :int =
(* INV: n! = a * i! *)
let rec loop (i:int) (a:int) :int =
  if i=0 then a
  else loop (i-1) (a*i)
in loop n 1
```

- Initially $i = n$ and $a_n = 1$, so $a_n \times n! = n!$ is true.
- Assume $a_i \times i! = n!$ at iteration i. Then at iteration $i-1$, $a_{i-1} = a_i \times i$, so $a_{i-1} \times (i-1) = n!$.
- At termination $i = 1$ so $a_1 \times 1 = n!$.

Using invariants to construct functions

We can also use the invariant $a_i \times i! = n!$ to determine the arguments to the recursive call.

```
let fact (n:int) :int =
  (* INV: a * i! = n! *)
  let rec loop (i:int) (a:int) :int =
    if i=0 then a
    else loop (i-1) _____
  in loop n 1
```

Exercise

Transform the following function to tail-recursive form.

```
let rec pow (x:int) (n:int) :int =
  if n=0 then 1
  else x * pow x (n-1)
```

What is the recursion invariant?

$$a \times x^i = x^n$$

Tail recursive Fibonnacci

```
let rec fib (n:int) :int =
  if n=0 then 0
  else if n=1 then 1
  else fib (n-1) + fib (n-2)
```

- ★ We need two accumulators a and b to carry forward the values computed by the two recursive calls.
- ★ Write down the skeleton of the tail recursive fibonnacci.
- ★ Write the invariants for a and b using the formula,

$$fib\ n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

Tail recursive Fibonnacci skeleton

```
let fib (n:int) :int =
  let rec loop i a b =
    if i=0 then a
    else loop (i-1) _____ _____ in
  loop n 0 1
```

Fill in the blanks using the invariant.