Programming Abstractions

Lecture 6: Accumulator-passing style

Stephen Checkoway

Loops and efficiency

Compare a C (or Java) function to compute the factorial

```
int fact(int n) {
  int product = 1;
  while (n > 0) {
    product *= n;
    n -= 1;
  }
  return product;
}
```

to our recursive Racket implementation

```
(define (fact n)
  (if (<= n 1)
      1
      (* n
         (fact (- n 1)))))
```

How do these differ?

In C, just one function call

In Racket, `(fact 10)` makes 10 calls to fact (the original one and then nine more)

Loops and efficiency

To be efficient, Racket internally converts all **tail-recursions** into loops

A function is tail-recursive if the last thing it does is to recurse and return the result of that recursion

Example:

```
(define (foo x y)
  (if (zero? x)
    y
    (foo (sub1 x) (+ x y))))
```

When the condition is satisfied, `y` is returned, otherwise `foo` is called again with some different parameters and that value is returned

Our factorial is *not* tail recursive

```
(define (fact n)
  (if (<= n 1)
      1
      (* n
         (fact (- n 1)))))
```

The last thing fact does is perform a multiplication; the recursion happens before the multiplication

Our factorial is *not* tail recursive

Given `(fact 4)`, we end up with

```
(fact 4) => (* 4 (fact 3))
         => (* 4 (* 3 (fact 2)))
         => (* 4 (* 3 (* 2 (fact 1))))
         => (* 4 (* 3 (* 2 1)))
         => (* 4 (* 3 2))
         => (* 4 6)
         => 24
```

We can see this in DrRacket

Solution: Use an accumulator

(Accumulator-passing style isn't the real name of this technique)

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))
(define (fact2 n)
  (fact-a n 1))
```

Four things to notice

- We defined a recursive helper function that takes an additional param
- We provide an initial value for the accumulator in `fact2`'s call to `fact-a`
- The base case returns the accumulator
- `fact-a` is tail-recursive

fact2 is tail-recursive

```
(fact2 4) => (fact-a 4 1)
          => (fact-a 3 4)
          => (fact-a 2 12)
          => (fact-a 1 24)
          => 24
```

So how does this become a loop?

Use variables for the parameters and update them each time through the loop

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))
```

becomes (pseudocode)

```
def fact-a(n, acc):
  loop:
    if n <= 1:
      return acc
    n, acc = n - 1, n * acc
```

Is this procedure tail recursive?

```
(define (length lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (length (rest lst)))]))
```

A. Yes

B. No

C. It depends on how long the list is

Is this procedure tail recursive?

```
; Return the nth element of lst
(define (list-ref lst n)
  (cond [(empty? lst) (error 'list-ref "List too short")]
        [(zero? n) (first lst)]
        [else (list-ref (rest lst) (sub1 n))]))
```

A. Yes

B. No

C. I have no idea!

Two strategies for tail recursive procedures

Accumulator-passing style with one or more accumulator parameters

- Usually, the procedure we really want doesn't have these parameters
- Use helper functions

Continuation-passing style

- This uses something called *continuations* which we'll talk about later in the semester

Let's write some tail-recursion procedures

`(sum lst)` — Add all the numbers in the `lst`

`(maximum lst)` — Find the maximum value in a nonempty list

`(reverse lst)` — Reverses the list `lst`

`(remove* x lst)` — Remove all instances of `x` from `lst`

- If we use `letrec` to define `remove*-a`, then we don't need to pass `x` to `remove*-a`

`(remove x lst)` — Remove the first instance of `x` from `lst`

- We can use `letrec` here as well