CS342: Organization of Prog. Languages

Topic 11: Tail Calls

- Tail Recursion
- Tail Call Example
- Tail Recursion Elimination
- Using Tail Recursion
- Scheme Do Loops as Tail Recursion
- Other Sequencing with Tail Calls

Tail Recursion

- Which is better, the loop version or the recursive version of `factorial`?
- It turns out that programs which can be written as loops, can also be written in an efficient recursive form, using "Tail Calls", or "Tail Recursion".
- The Scheme language definition *requires* that tail calls have this efficient implementation

Tail Call Example

- Consider the following example:

```
(define (f n)  (g (* 3 n)))
(define (g n)  (+ n 1))
(f 7)
```

- Step 1. Enter `f`. Set `f`'s `n = 7`.
- Step 2. Eval (* 3 n) to get `21`.
- Step 3. Enter `g`. Set `g`'s `n = 21`.
- Step 4. Eval `(+ n 1)` to get 22.
  Notice that there is absolutely `g` *has to do with this value belfore it, itself, returns the value to* `f`*.*
- Step 5. Return 22 from `g` to `f`.
  Notice that there is absolutely `f` *has to do with this value belfore it, itself, returns.*
- Step 6. Return 22 from `f` to `f`'s caller.

Tail Recursion Elimination

- Consider the following tail recursive C program:

```
int
my_strlen_plus(char *s, int n)
{
     if (*s == 0) return n;
     return my_strlen_plus(s + 1, n + 1);
}
int my_strlen(char *s) { return my_strlen_plus(s, 0); }
```

  The calls to `my_strlen_plus` are all tail recursive.

- Rewrite `my_strlen_plus` to re-use the local environment rather than stacking a recursive call:

```
int
my_strlen_plus(char *s, int n)
{
restart_me:
     if (*s == 0) return n;
     s = s + 1;
     n = n + 1;
     goto restart_me;
}
```

- In C there is no elegant way to efficiently to handle tail calls between different functions.
- Reminder: Scheme requires that all tail-recursive calls be handled efficiently.

Using Tail Recursion

- Consider the following recursive function:

```
;; General recursion

(define (my-length L)
    (if (null? L) 0 (+ 1 (my-length (cdr L)))) )
```

  Note that the result of the recursive call to `my-length` has 1 added to it.

- Let's rewrite my-length using tail recursion so it does not accumulate stack frames.
  It then can in principle be as efficient as a C loop when compiled:

```
;; New -- tail recursion

(define (better-length L0)
   (letrec
       ((f (lambda (L n)
              (if (null? L) n (f (cdr L) (+ 1 n)))) ))
     (f L0 0) ))
```

Scheme Do Loops as Tail Recursion

- Recall the form of a `do` loop in Scheme:

```
(do  ((var1 init1 step1) ...) (test fini1 ...)
     body1 body2 ...)
```

- This form is equivalent to a tail recursive function application where:
  - The loop body becomes recursive function.
  - The loop control variables as parameters.

```
(letrec ((privat-loop-fn
          (lambda (var1 ...)
             (if test
                 (begin fini1 ...)
                 (begin
                    body1
                    body2 ...
                    (privat-loop-fn step1 ...) ))) ))

        (privat-loop-fn init1 ...) )
```

- Example:

  Loop:

```
(define factorial (lambda (n)
   (let ((prod 1))
      (do ((i 1 (+ 1 i)))
          ((> i n) prod)
          (set! prod (* i prod)) ))))
```

  Equivalent tail-recursion:

```
(define factorial (lambda (n)
   (let ((prod 1))
      (letrec ((factorial-loop-fn
          (lambda (i)
             (if (> i n)
                 (begin prod)
                 (begin
                    (set! prod (* i prod))
                    (factorial-loop-fn (+ i 1)) ))) ))
       (factorial-loop-fn 1) ))))
```

Other Sequencing with Tail Calls

- Begin by considering how the control structure can be described in terms of gotos, e.g.:

```
int f(int a, int b) {
        int i, tot = 0;
        for (i = a; i <=b; i++) tot = tot + i;
        return tot;
}
```

This is equivalent to a program with gotos as:

```
int f(int a, int b) {
       int i, tot = 0;

entry:  i = a;
again:  if (i > b) goto done;
       tot = tot + i;
       i++;
       goto again;
done:   return tot;
}
```

Other Sequencing with Tail Calls (cont'd)

- Now convert each block into a nullary function.
- The `goto`s become tail calls.

```
(define (f a b)
   (let ((i #f) (tot 0))
      (letrec (
          (entry (lambda ()
            (set! i a)
            (again) ))          ; Tail call

          (again (lambda ()
            (if (> i b)
              (done)            ; Tail call
              (begin
                (set! tot (+ tot i))
                (set! i (+ i 1))
                (again) ))))    ; Tail call

          (done (lambda ()
            tot )) )            ; Return result

        (entry)) ))             ; Tail call
```