61A LECTURE 22 – TAIL CALLS, ITERATORS

Steven Tang and Eric Tzeng

July 30, 2013

Announcements

- Homework 12 due Thursday, not Monday.
  - Take the time to get started on the project instead!

Almost done with Scheme

- Scheme is a lot simpler than Python, so it has a lot less features than Python does
- Today we're talking about a feature that Scheme has that Python doesn't!

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames

Time	Space
$\Theta(n)$	$\Theta(n)$

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Time	Space
$\Theta(n)$	$\Theta(1)$

```
def factorial(n):
    total = 1
    while n > 0:
        n, total = n - 1, total * n
    return total
```

Iteration and Recursion

Reminder: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

```
def factorial(n):
    total = 1
    while n > 0:
        n, total = n - 1, total * n
    return total
```

Local names become…

Parameters in a recursive function

```
def factorial(n, total):
    if n == 0:
        return total
    return factorial(n - 1, total * n)
```

But this converted version still uses linear space in Python

Tail Recursion

From the *Revised[7] Report on the Algorithmic Language Scheme*:

"Implementations of Scheme are required to be **properly tail-recursive**. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

```
(define (factorial n total)
  (if (= n 0) total
      (factorial (- n 1)
                 (* total n))))
```

```
def factorial(n, total):
    if n == 0:
        return total
    return factorial(n - 1, total * n)
```

Tail Calls

A procedure call that has not yet returned is *active*. Some procedure calls are *tail calls*. A Scheme interpreter should support an unbounded number of active tail calls.

A tail call is a call expression in a *tail context*:

- The last body sub-expression in a **lambda** expression
- Sub-expressions 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and** or **or**
- The last sub-expression in a tail context **begin**

```
(define (factorial n total)
  (if (= n 0) total
      (factorial (- n 1)
                 (* total n)) ) )
```

Example: Length of a List

```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s))) ) ) )
```

Not a tail context

A call expression is not a tail call if more computation is still required in the calling procedure.

Linear recursions can often be rewritten to use tail calls.

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n)) ) )
  (length-iter s 0) )
```

Recursive call is a tail call

Eval with Tail Call Optimization

The return value of the tail call is the return value of the current procedure call.

Therefore, tail calls shouldn't increase the environment size.

In the interpreter, recursive calls to **`scheme_eval`** for tail calls must instead be expressed iteratively.

Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- **Cond** expr'n: `(cond (<p1> <e1>) ... (<pn> <en>) (else <e>))`

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: `<consequent>` or `<alternative>`

do_if_form

- Evaluate that sub-expression in place of the whole expression.

scheme_eval

Evaluation of the tail context does not require a recursive call.

E.g., replace `(if false 1 (+ 2 3))` with `(+ 2 3)` and iterate.

Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
      (reduce procedure
              (cdr s)
              (procedure start (car s)) ) ) )
```

Recursive call is a tail call.

Other calls are not; constant space depends on **`procedure`**.

```
(reduce * '(3 4 5) 2)                                   120

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))        (5 4 3 2)
```

Example: Map

```
(define (map procedure s)
  (define (map-iter procedure s m)
    (if (null? s) m
        (map-iter procedure
                  (cdr s)
                  (cons (procedure (car s)) m))))
  (reverse (map-iter procedure s nil)))

(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s) r
        (reverse-iter (cdr s)
                      (cons (car s) r))))
  (reverse-iter s nil))
```

Break!

Guido van Rossum
"Benevolent Dictator for Life"

- Remember this guy?
- Guido on tail call optimization:
  - "Personally, I think it is a fine feature for some languages, but I don't think it fits Python"
- Why do you think this is?

Our Sequence Abstraction

Recall our previous sequence interface:

- A sequence has a finite, known length
- A sequence allows element selection for any element

In the cases we've seen so far, satisfying the sequence interface requires storing the entire sequence in a computer's memory

Problems?

- Infinite sequences - primes, positive integers
- Really large sequences - all Twitter tweets, votes in a presidential election

The Sequence of Primes

Think about the sequence of prime numbers:

- What's the first one?
- The next one?
- The next one?
- How about the next two?
- How about the 105th prime?
    - Our sequence abstraction would give an instant answer

Implicit Sequences

- We compute each of the elements on demand
- Don’t explicitly store each element
- Called an **implicit sequence**

A Python Example

Example: The `range` class represents a regular sequence of integers

- The range is represented by three values: *start*, *end*, and *step*
- The length and elements are computed on demand
- Constant space for arbitrarily long sequences

$$length = max\left(\left\lceil \frac{end - start}{step} \right\rceil, 0\right)$$

$$elem(k) = start + k \cdot step \quad (\text{for } k \in [0, length))$$

A Range Class

```
class Range(object):
    def __init__(self, start, end=None, step=1):
        if end is None:
            start, end = 0, start
        self.start = start
        self.end = end
        self.step = step

    def __len__(self):
        return max(0, ceil((self.end - self.start) /
                           self.step))

    def __getitem__(self, k):
        if k < 0:
            k = len(self) + k
        if k < 0 or k >= len(self):
            raise IndexError('index out of range')
        return self.start + k * self.step
```

The Iterator Interface

An iterator is an object that can provide the next element of a (possibly implicit) sequence

The iterator interface has two methods:

- **`__iter__(self)`** returns an equivalent iterator
- **`__next__(self)`** returns the next element in the sequence
  - If no next, raises **`StopIteration`** exception

There are also built in functions **`next`** and **`iter`** that call the corresponding method on their argument.

RangeIter

```
class RangeIter(object):
    def __init__(self, start, end, step):
        self.current = start
        self.end = end
        self.step = step
        self.sign = 1 if step > 0 else -1

    def __next__(self):
        if self.current * self.sign >= self.end * self.sign:
            raise StopIteration
        result = self.current
        self.current += self.step
        return result

    def __iter__(self):
        return self
```

Fibonacci

```
class FibIter(object):
    def __init__(self):
        self.prev = -1
        self.current = 1

    def __next__(self):
        self.prev, self.current = (self.current,
                                   self.prev + self.current)
        return self.current

    def __iter__(self):
        return self
```

The For Statement

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header **<expression>**, which yields an iterable object.
2. For each element in that sequence, in order:
   A. Bind **<name>** to that element in the first frame of the current environment
   B. Execute the **<suite>**

An iterable object has a method **__iter__** that returns an iterator

```
>>> nums, sum = [1, 2, 3], 0
>>> for item in nums:
        sum += item
>>> sum
6
```

```
>>> nums, sum = [1, 2, 3], 0
>>> items = nums.__iter__()
>>> try:
        while True:
            item = items.__next__()
            sum += item
    except StopIteration:
        pass
>>> sum
6
```