MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes – Sept. 12, 1996

Models of Computation

We need models to understand the evolution of a computational process under the direction of procedures. Our initial model is an informal substitution model.

The particular questions that our model is designed to address are ones about resources:

- How many steps are performed in the execution of a process?
- How much space is required to execute a process?

Here are the rules for our model:

- "Elementary expressions" are left alone: Elementary expressions are
  - numerals
  - initial names of primitive procedures
  - lambda expressions, naming procedures
- A nanme bound by DEFINE: Rewrite the name as the value it is associated with by the definition
- IF: If the evaluation of the predicate expression terminates in a non-false value
  - then rewrite the IF expression as the value of the consequent expression,
  - otherwise, rewrite the IF expression as the value of the alternative expression.
- Combination – (<operator> <operand1> ... <operandn>):
  - Evaluate the operator expression to get the procedure, and evaluate the operand expressions to get the arguments,
  - If the operator names a primitive procedure, do whatever magic the primitive procedure does.
  - If the operator names a compound procedure, evaluate the body of the compound procedure with the arguments substituted for the formal parameters in the body. Count internal definitions as part of the body – watch out for substitution bugs here!

Here's an example evaluation

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

and here is the substitution rewrites:

```
(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f  1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #f 1 (* 3 (fact (- 2 1)))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 (if#t 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 1))
(* 3 2)
6
```

There are other possible models, such as normal order evaluation.

Now, let's look at some examples of common procedures, and how the substitution model let's us examine the evolution of such programming gambits.

Exponentiation – first try

```
(define exp-1
  (lambda (a b)
     (if (= b 0)
         1
         (* a (exp-1 a (- b 1))))))
```

This is a linear recursive process.

Exponentiation – second try

```
(define exp-2
  (lambda (a b)
     (define exp-iter
       (lambda (a b ans)
          (if (= b 0)
              ans
              (exp-iter a (- b 1) (* a ans)))))
     (exp-iter a b 1)))
```

This is a linear iterative process.

To capture the differences between these processes, we use the idea of Orders of Growth. Given some resource, such as space or time, the order of growth of a process describes how quickly the demands for that resource increase as we increase the size of the problem.

More formally, if $R(n)$ denotes the amount of resource needed to solve a problem of size n and we can find constants k_1 and k_2 and a mathematical function $f(n)$ such that

$$k_1 f(n) \leq R(n) \leq k_2 f(n)$$

for large n, then $R(n)$ is said to have order of growth $f(n)$ which we write as $\Theta(f(n))$.

In our case `exp-1` has order of growth $\Theta(n)$ in both space and time, while `exp-2` has order of growth $\Theta(n)$ in time, but $\Theta(1)$ in space.

Example – Towers of Hanoi – Tree recursion

To solve this problem, we use the technique of wishful thinking, which says

- decide how to reduce the general problem to a simpler one
- pretend we can solve the simpler one
- use the simpler solution to construct a solution to the general one

Here is a solution based on this idea:

```
(define move-tower
  (lambda (size from to extra)
     (cond ((= size 0) true)
           (else (move-tower (- size 1) from extra to)
                 (print-move from to)
                 (move-tower (- size 1) extra to from)))))

(define print-move
   (lambda (from to)
     (write-line ``Move top disk from ``)
     (write-line from)
     (write-line `` to ``)
     (write-line to)))
```

The orders of growth here are $\Theta(2^n)$ in time and $\Theta(n)$ in space, and we call such procedures exponential.

Finally, here is a third way of writing `exp`

```
(define fast-exp-1
   (lambda (a b)
     (cond ((= b 1) a)
           ((even? b) (fast-exp-1 (* a a) (/ b 2)))
           (else (* a (fast-exp-1 a (- b 1)))))))
```

This procedure has logarithmic growth, $\Theta(\log(n))$, both in space and in time.