

Lecture 8: Making iterative processes, the Church-Turing thesis

An iterative version of length

Recall our previous procedure to find the length of a list:

```
(define our-length
  (lambda (ls)
    (if (null? ls)
        0
        (+ 1 (our-length (cdr ls))))))
```

This generates a recursive process because there is computation (adding 1) that takes place after the recursive call is evaluated. To convert this into a procedure that generates an iterative process, we add another parameter, which will give us somewhere to accumulate partial results. We define:

```
(define new-length
  (lambda (ls count)
    (if (null? ls)
        count
        (new-length (cdr ls) (+ count 1)))))
```

The value of the recursive call is the value of the whole procedure, so this procedure generates an iterative process. The recursive step is still with `(cdr ls)`, but also with `(+ count 1)`. A trace of this function may help clarify how it works:

```
> (new-length '(a b c) 0)
CALL new-length (a b c) 0
CALL new-length (b c) 1
CALL new-length (c) 2
CALL new-length () 3
RETN new-length 3
RETN new-length 3
RETN new-length 3
RETN new-length 3
3
```

In each successive call, the length of the list is one less, but the value of the other parameter is one more. In each call, the sum of `count` and the length of `ls` remains the same, namely 3. This is an **invariant** of the procedure. When the list becomes empty, the value of the other parameter is the length of the original list, which is why `count` is returned in the base case. One indicator that this is an iterative process is that all the return values are the same; if they were different, then computation would have to be going on after the recursive call returns.

But `(new-length ls count)` doesn't quite solve the original problem, of a procedure of **one** argument to find the length of a list. We write a procedure whose whole job is to call `new-length` with the correct initial value of the additional argument:

```
(define length-it
  (lambda (ls)
    (new-length ls 0)))
```

These two procedures together give a solution to calculating the length of a list via an iterative process.

An iterative process to reverse a list

We write a procedure to reverse a list that generates an iterative process. In this case, the additional argument is a list, which gives us a place to accumulate the partial results of reversing the input list. The two-argument procedure:

```
(define new-reverse
  (lambda (ls1 reversed-elements)
    (if (null? ls1)
        reversed-elements
        (new-reverse (cdr ls1) (cons (car ls1) reversed-elements)))))
```

There is no computation after the return of the recursive call to `new-reverse`, so this procedure generates an iterative process. Here is a trace of it:

```
> (new-reverse '(a b c) '())
CALL new-reverse (a b c) ()
CALL new-reverse (b c) (a)
CALL new-reverse (c) (b a)
CALL new-reverse () (c b a)
RETN new-reverse (c b a)
RETN new-reverse (c b a)
RETN new-reverse (c b a)
RETN new-reverse (c b a)
(c b a)
```

Each recursive call moves the first element of the first list to be the first element of the second list. When the first list becomes empty, the second list holds the original elements in reverse order. The **invariant**, or unchanging value, in all these calls is

```
(append (reverse ls1) reversed-elements) => (c b a)
```

In general, this expression evaluates to the reverse of the original list in every call. In the base case, the list `ls1` is empty, so the value of `reversed-elements` must be the reverse of the original list. This is why the base case can correctly return `reversed-elements`. Note again the unchanging return values.

Once again, we need another procedure to call this one with the correct value of the auxiliary parameter:

```
(define reverse-it
  (lambda (ls)
    (new-reverse ls '())))
```

In the example of `length-it`, the correct initial value is 0, but here it is the empty list. These two procedures together compute the reverse of a list and generate an iterative process.

An iterative process to simulate a Turing machine

Another example of a procedure that generates an iterative process is the Turing machine simulator from homework 2, which is reproduced here.

```
(define simulate
```

```
(lambda (mach config)
  (display config) (newline)
  (if (halted? mach config)
      'halted
      (simulate mach (next-config mach config))))
```

Each call to the procedure calls itself recursively once, and the value is returned as the value of the computation. In fact, if there is a value returned, we know what it is, namely, the symbol `halted`. The computation need not return a value at all (if the Turing machine never halts.) In fact, we are primarily interested in the side-effects: the successive configurations printed out by the calls to `display` and `newline`.

Common to each of these examples is that the main computation occurs in the expressions for the **arguments** to the recursive call (adding 1 to `count`, putting the next element at the front of `reversed-elements`, or computing the next configuration from the present one.)

Constructors and selectors

Homework 2 defines some constructors and selectors specific to Turing machines. For example, for Turing machine instructions we have the following.

```
(define make-i
  (lambda (state symbol next-symbol next-state direction)
    (list state symbol next-symbol next-state direction)))

(define i-state
  (lambda (inst) (list-ref inst 0)))

(define i-symbol
  (lambda (inst) (list-ref inst 1)))

(define i-next-symbol
  (lambda (inst) (list-ref inst 2)))

(define i-next-state
  (lambda (inst) (list-ref inst 3)))

(define i-direction
  (lambda (inst) (list-ref inst 4)))
```

The constructor, `make-i` takes the five elements of a Turing machine instruction, and puts them in a data structure from which they can be retrieved (in this case, just a list) and returns the value of that data structure. The selectors take an instance of that data structure and return the value of the selected component. Thus we might write

```
(define inst (make-i 2 'a 'c 4 'r))
```

To select a component of the instruction, say the next state, we use the appropriate selector:

```
(i-next-state inst) => 4
```

Constructors and selectors are also defined for tapes and configurations. Why do we bother with this, when we could just use the built-in Scheme constructors and selectors (for example, `cons`, `car`, and `cdr` for lists)

and combinations of them? One reason is to make the structure of the code more reflective of the way we think about the problem. For example, `(i-next-state inst)` conveys more meaning to the person reading the code than `(caddr inst)` or even `(list-ref inst 3)`. Another reason is to modularize the code, so that a change in the data structure to represent a certain object need only affect the relevant constructors and selectors rather than requiring changes scattered through the program.

The Church-Turing thesis

The Church-Turing Thesis

The Church-Turing thesis states that

Every computable function is computable by a Turing machine.

Why is this a thesis rather than a theorem? Because in this statement, the first instance of “computable” refers to our intuitive meaning of this word, and the second instance of “computable” refers to a specific formal definition of what it means for a Turing machine to compute a function. The “thesis” then says that the formal notion of computation by a Turing machine satisfactorily captures what we intuitively mean when we say a function is computable.

As Turing states in his paper, the evidence for this thesis consists of (a) intuitive arguments about what we mean by computation, (b) the provable fact that a number of apparently very different computational systems compute exactly the same set of functions as Turing machines, and (c) examples of interesting, powerful functions that are computable by a Turing machine (or equivalent computational system.)

To see what (b) means in more detail, consider the sets of functions from the natural numbers to the natural numbers computable by (1) Turing machines, (2) lambda expressions, (3) Markov algorithms, (4) general recursive functions, (5) Post correspondence systems, and (6) random access machines. (Yes, we’ve only looked at Turing machines in any detail. The lambda calculus is the theoretical basis of Lisp and Scheme, and random access machines are a simplified model of modern computer architecture.) The remarkable thing is that all six of these sets of functions turn out to be the same. The theorems that prove this are simulation theorems, which give, for example, a method of simulating a Turing machine using a lambda calculus expression. In fact, the thesis as expressed by Church has lambda expressions instead of Turing machines.

A programming language or system is said to be Turing-complete if it can compute all and only the functions computable by a Turing machine. In the homework you are asked to write a Turing machine simulator in Scheme; a proof of the correctness of your simulator would be half of a proof of the Turing-completeness of an idealized version Scheme (with no memory limitations). The other half would be a proof that a Turing machine could simulate the idealized version of Scheme. You’ll be relieved to know that I haven’t asked you to write a Turing machine to simulate Scheme for homework 2.

Should you believe the Church-Turing thesis? It is generally accepted by computer scientists. However, it is conceivable that some physical phenomenon (black holes? time travel?) might imply that our current understanding of computational limitations is flawed, which in turn might suggest a theoretical reformulation that changes the class of computable functions.