TABLES, VECTORS 16

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 20, 2010

1 Tables

1.1 Tables

Tables. We're now ready to understand how to implement the put and get procedures that A&S used at the end of chapter 2. A table is a list of key-value pairs, with an extra element at the front just so that adding the first entry to the table will be no diffferent from adding later entries. (That is, even in an "empty" table we have a pair to set-cdr!)

```
;;;;; In file cs61a/lectures/3.3/table.scm
(define (get key)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        #f
        (cdr record))))
(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr! the-table (cons (cons key value)
                                  (cdr the-table)))
        (set-cdr! record value)))
    'ok)
(define the-table (list '*table*))
```

Assoc is in the book:

```
(define (assoc key records)
  (cond ((null? records) #f)
        ((equal? key (caar records))
         (car records))
```

```
        (else (assoc key (cdr records))) ))
```

In chapter 2, A&S provided a single, global table, but we can generalize this in the usual way by taking an extra argument for which table to use. That's how `lookup` and `insert!` work. One little detail that always confuses people is why, in creating two-dimensional tables, we don't need a `*table*` header on each of the subtables. The point is that `lookup` and `insert!` don't pay any attention to the `car` of that header pair; all they need is to represent a table by some pair whose `cdr` points to the actual list of key-value pairs. In a subtable, the key-value pair from the top-level table plays that role. That is, the entire subtable is a value of some key-value pair in the main table. What it means to be "the value of a key-value pair" is to be the `cdr` of that pair. So we can think of that pair as the header pair for the subtable.

2 Vectors

So far we have seen one primitive data aggregation mechanism: the pair. We use linked pairs to represent sequences in the form of lists.

The list suffers from one important weakness: Finding the nth element of a list takes time $O(n)$ because you have to call `cdr` $n-1$ times. Scheme, like most programming languages, also provides a primitive aggregation mechanism without this weakness. In Scheme it's called a vector; in many other languages it's called an array, but it's the same idea. Accessing the nth element of a vector takes $O(1)$ time.

2.1 Vector Primitives

Some of the procedures for vectors are exact analogs to procedures for lists:

```
(vector a b c d ...)     (list a b c d ...)
(vector-ref vec n)       (list-ref lst n)
(vector-length vec)      (length lst)
```

Most notably, the selector for vectors, `vector-ref`, is just like the selector for lists (except that it's faster). What about constructors? There's a `vector` procedure, just like the `list` procedure, that's good for situations in which you know exactly how many elements the sequence will have, and all of the element values, all at once. But there are no vector analogs to the list constructors `cons` and `append`, which are useful for extending lists. In particular, `cons` is the workhorse of recursive list processing procedures; we'll see that vector processing is done quite differently.

The weakness of vectors is that they can't be extended. You have to know the length of the vector when you create it. So instead of `cons` and `append` we have `(make-vector len)` which creates a vector of length `len`, in which the element values are unspecified. (You then use mutation, discussed below, to fill in the desired values.) Alternatively, if you want to create a vector in which every element has the same initial value, you can say `(make-vector len value)`.

Because vectors are created all at once, rather than one element at a time, mutation is crucial to any useful vector program. The primitive mutator for vectors is `(vector-set! vec n value)` This procedure is comparable to `set-car!` and `set-cdr!` for pairs. (It's interesting to note that Scheme doesn't provide a mutator for the nth element of a list. This is because most list processing is done using functional programming style, and pair mutation is mainly for special cases such as tables.) The printed format of a vector is `#(a b c d)`

You can quote this to include a constant vector in a program. (Note: In STk, vectors are self-evaluating, so you can omit the quotation mark, but this is a nonstandard extension to Scheme.) Scheme also provides functions `list->vector` and `vector->list` that let you convert between the two sequence implementations.

2.2 Writing Vector Programs

Let's write a mapping function for vectors; it will take a function and a vector as arguments, and return a vector. For reference, here's the map function on a list.

```
(define (map fn lst)
  (if (null? lst)
      '()
      (cons (fn (car lst))
            (map fn (cdr lst)))))
```

To do the same task for vectors, we must first create a new vector of the same length as the argument vector, then fill in the values using mutation:

```
;;;;; In file cs61a/lectures/vector.scm
(define (vector-map fn vec)
  (define (loop newvec n)
    (if (< n 0)
        newvec
        (begin (vector-set! newvec n (fn (vector-ref vec n)))
               (loop newvec (- n 1)))))
  (loop (make-vector (vector-length vec))
        (- (vector-length vec) 1)))
```

This is a lot more complicated! It requires a helper procedure, and an extra index variable, `n`, to keep track of the element number within the vector. By contrast, the list version of map never actually knows how long its argument list is.

2.3 `vector-map!`

Write a procedure that is the same as the above, but changes it in-place. In other words, without returning a new vector.

2.4 `vector-filter!`

Write a procedure vector-filter, but changes it in-place. In other words, without returning a new vector. Since you will have less items than when you started, replace the rest of the list with `#f`.

2.5 Comparison with Lists

Of course, if we wanted, we could write our own equivalent to `cons` for vectors:

```
;;;;; In file cs61a/lectures/vector.scm
(define (vector-cons value vec)
  (define (loop newvec n)
    (if (= n 0)
        (begin (vector-set! newvec n value) newvec)
        (begin (vector-set! newvec n (vector-ref vec (- n 1)))
               (loop newvec (- n 1)))))
```

```
(loop (make-vector (+ (vector-length vec) 1))
      (vector-length vec)))
```

If we wrote similar procedures `vector-car` and `vector-cdr`, we could then write `vector-map` in a style exactly like `map`. But this would be a bad idea, because our `vector-cons` requires $O(n)$ time to copy the elements from the old vector to the new one.

Operation	Lists	Vectors
nth Element	`list-ref` $\Theta(n)$	`vector-ref` $\Theta(1)$
Add New Element	`cons` $\Theta(1)$	`vector-cons` $\Theta(n)$

This is why there isn't one single best way to represent sequences. Lists are faster (and allow for cleaner code) at adding elements, but vectors are faster at selecting arbitrary elements. (Note, though, that if you want to select all the elements of a sequence, one after another, then lists are just as fast as arrays. It's only when you want to jump around within the sequence that arrays are faster.)

2.6 Example: Shuffling

Suppose we want to shuffle a deck of cards — we want to reorder the cards randomly. We'll look at three solutions to this problem.

First, here's a solution using functional programming with lists. Because we aren't allowing mutation of pairs, this version does a lot of recopying:

```
;;;;; In file cs61a/lectures/vector.scm
(define (shuffle1 lst)
  (define (loop in out n)
    (if (= n 0)
        (cons (car in) (shuffle1 (append (cdr in) out)))
        (loop (cdr in) (cons (car in) out) (- n 1))))
  (if (null? lst)
      '()
      (loop lst '() (random (length lst)))))
```

This is a case in which functional programming has few virtues. The code is hard to read, and it takes $\Theta(n^2)$ time to shuffle a list of length n. (There are n recursive calls to shuffle1, each of which calls the $\Theta(n)$ primitives `append` and `length` as well as $\Theta(n)$ calls to the helper function `loop`.) We can improve things using list mutation. Any list-based solution will still be $\Theta(n^2)$, because it takes $\Theta(n)$ time to find one element at a randomly chosen position, and we have to do that n times. But we can improve the constant factor by avoiding the copying of pairs that `append` does in the first version:

```
;;;;; In file cs61a/lectures/vector.scm
(define (shuffle2! lst)
  (if (null? lst)
      '()
      (let ((index (random (length lst))))
        (let ((pair ((repeated cdr index) lst))
              (temp (car lst)))
          (set-car! lst (car pair))
          (set-car! pair temp)
          (shuffle2! (cdr lst))
          lst)))))
```

(Note: This could be improved still further by calling length only once, and using a helper procedure to subtract one from the length in each recursive call. But that would make the code more complicated, so I'm

not bothering. You can take it as an exercise if you're interested.)

Vectors allow a more dramatic speedup, because finding each element takes $\Theta(1)$ instead of $\Theta(n)$:

```
;;;;; In file cs61a/lectures/vector.scm
(define (shuffle3! vec)
  (define (loop n)
    (if (= n 0)
        vec
        (let ((index (random n))
              (temp (vector-ref vec (- n 1))))
          (vector-set! vec (- n 1) (vector-ref vec index))
          (vector-set! vec index temp)
          (loop (- n 1)))))
  (loop (vector-length vec)))
```

The total time for this version is $\Theta(n)$, because it makes n recursive calls, each of which takes constant time.

2.7 How Vectors Work

One handwavy paragraph on why vectors have the performance they do: A pair is two pointers attached to each other in a single block of memory. A vector is similar, but it's a block of n pointers for an arbitrary (but fixed) number n. Since a vector is one contiguous block of memory, if you know the address of the beginning of the block, you can just add k to find the address of the kth element. The downside is that in order to get all the elements in a single block of memory, you have to allocate the block all at once. If you don't understand that, don't worry about it.