Notes on Project 5

December 8, 2012

About "recursively memoize oper," and its type. In the problem about maximum nonadjacent subsequences, you will first construct a "knowledge extension operator" called `mnas_oper`.

This operator depends on the sequence s that it is supposed to work on, and it builds an operator f (meaning, a *function* f) that will extend knowledge about the sums of maximum nonadjacent subsequences of s. The result f is a function that takes an index i and a "current knowledge" operator g. Then $f(i, g)$ will return the maximum sum for a nonadjacent subsequence of s that uses at must the first i positions of s. It should give the right answers up to some maximum j *assuming* that g gives the right answers up to maximum $j - 1$.

So $f(i, g)$ *extends* the knowledge contained in g.

Notice that the "type" of g is $\mathsf{Int} \to \mathsf{Int}$. That means, given an integer argument i, $g(i)$ is an integer, namely the sum of the maximum nonadjacent subsequence of s that uses at most the first i entries from s. (Assume that s is a sequence of integers, so that the sum is an integer too.) We can write this:

$$g\colon \mathsf{Int} \to \mathsf{Int}$$

That means that the type of f will be:

$$f\colon \mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}.$$

This says that if f is given an integer i and an argument g with type $g\colon \mathsf{Int} \to \mathsf{Int}$, then $f(i, g)$ should be an integer.

So what is the type of `mnas_oper`? Well, it is a function that takes as argument an integer array, and it returns f. Let us write the type of an integer array as $\mathsf{Int\ array}$. Since we already know the type of f, we can write the type of `mnas_oper` as:

$$\texttt{mnas\_oper}\colon \mathsf{Int\ array} \to (\mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}).$$

We are now ready to figure out the type of `recursively_memoize_oper`. It can take as argument a function like f, which is called `o` in the max_nonadj.lua file. We know that f has type $f\colon \mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$. It returns a function `fn` which takes an argument that it gives as first argument to f, and `fn` returns the same answer that f returned. That means that the argument and return value of `fn` are the same as the first argument and the return value of f. So `fn`$\colon \mathsf{Int} \to \mathsf{Int}$.

Since `recursively_memoize_oper` takes an argument of type $f\colon \mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$ and returns a function of type $\mathsf{Int} \to \mathsf{Int}$, we have:

$$\texttt{recursively\_memoize\_oper}\colon (\mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}) \to (\mathsf{Int} \to \mathsf{Int}).$$

We can use this to explain how `mnas_max` works. Its definition says:

```
function mnas_max(s)
   return recursively_memoize_oper(mnas_oper(s))(#s)
end
```

It takes an argument s which is an integer array, i.e., $s\colon \mathsf{Int\ array}$. Thus,

$$\texttt{mnas\_oper(s)}\colon \mathsf{Int} \times (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$$

since this is the type of value that `mnas_oper` returns. Thus:

$$\texttt{recursively\_memoize\_oper(mnas\_oper(s))}\colon \mathsf{Int} \to \mathsf{Int}.$$

Now $\#s$ is an integer, namely the length of s. So:

$$\texttt{recursively\_memoize\_oper(mnas\_oper(s))(\#s)}\colon \mathsf{Int}.$$

This is exactly what we want: It means that `mnas_max` returns an integer, given any sequence (array) of integers.

Curiously, `recursively_memoize_oper` can also be applied to arguments of other types. This is the relevant story for our problem, however.

Knowledge extension operators and recurrences. The project write-up and the starter code file both use the word "recurrence." In both of these, they mean what we have been calling a knowledge extension operator. "Recurrence" is not a crazy thing to call a knowledge extension operator, since it does some work but does "recur" to its function argument g as needed.

However, I should emphasize that this has nothing to do with recurrences like the ones that the Master Theorem talks about, such as $T(n) = aT(n/b) + f(n)$.

Bellman-Ford and Finding the Shortest Paths. Suppose that you write your Bellman-Ford implementation, and call it on a graph g and starting node s. The code is supposed to give you back two tables. One is the table d whose entry for node t is the length of the shortest path $s \to^* t$. If the entry in d is nil, that means that there was no path from s to t[1].

The other is the table π, i.e. pi. $\pi[n]$ is the parent, i.e. predecessor, of n along some shortest path from s. So some shortest path is of the form $s \to^* \pi[t] \to t$; its last arc takes us from $\pi[t]$ to t. We can use π again to find out the earlier part of this path, as in:

$$s \to^* \pi[\pi[t]] \to \pi[t] \to t.$$

You can keep using this idea backward to get the whole path between s and t worked out.

[1] I'm using $s \to^* t$ to mean that we're allowed to use $\to$ zero or more times repeatedly to get to t.