14 Countdown, dynamic programming solutions

The previous lecture defined solvers for the Countdown problem which took the form

$$\begin{array}{lcll} \mathit{solutions}\ \mathit{ns}\ n & = & [(e,v) \mid & \mathit{ys} \leftarrow \mathit{subs}\ \mathit{ns}, \\ & & & \mathit{cs} \leftarrow \mathit{permutations}\ \mathit{ys}, \\ & & & (e,v) \leftarrow \mathit{results}\ \mathit{cs}, \\ & & & v == n] \end{array}$$

and where the *results* function was repeatedly refined. In each case the *results* function then decomposes *cs* into subseqences on which it builds expressions. Consequently the same work is being done repeatedly.

14.1 Tabulation for efficiency

The well-known Fibonacci sequence, the image of the natural numbers under

```
> fib :: Int -> Integer
> fib 0 = 0
> fib 1 = 1
> fib n = fib (n-1) + fib (n-2)
```

is most naturally specified by that function definition. However, this is an inefficient way of computing elements of the sequence.

Since $\mathit{fib}\ n \approx \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n$, and since the result is produced by adding 1 to itself (and to some zeroes) this function (read as a program) gives an exponentially slow way of calculating elements of the sequence.

Observe that in calculating $\mathit{fib}\ n$ there are about $2 \times \mathit{fib}\ n$ calls of *fib* but only n different calls, since each call has an argument less than n. It would be better to calculate each of these once, and then use their values as needed.

One way of understanding tabulation is first to extract the *kernel* of the recursion for *fib*.

```
> fibK :: (Int -> Integer) -> Int -> Integer
> fibK f 0 = 0
> fibK f 1 = 1
> fibK f n = f (n-1) + f (n-2)
```

This kernel function is not itself recursive: all previously recursive calls are replace by calls to the argument f, however $\mathit{fib}_K\ \mathit{fib} = \mathit{fib}$. That is, *fib* is a fixed point of fib_K, in the same way that 1 is a fixed point of $g\ x = (x^2+1)/2$, because $g\ 1 = 1$.

Define $\mathit{fix}\ k = k(\mathit{fix}\ k)$, then $\mathit{fix}\ k$ is clearly a fixed point of k, and indeed $\mathit{fix}\ \mathit{fib}_K$ is equal to *fib*. (However, $\mathit{fix}\ g$ is not 1, because g is strict and so $\perp$ is also a fixed point of g; $\mathit{fix}\ k$ is always the least-defined fixed point of k.)

Of course, $fix\ fib_K$ will not only implement the same function as fib, but also it calls fib_K exactly as many times as the original program called fib, and so has the same exponentially poor performance.

The *dynamic programming* strategy for evaluating a function such as fib is to construct a table of values of the function, and to implement the function by looking up entries in that table. The table itself is constructed by calling the kernel, but with the table-lookup implementation as an argument.

In the case of fib, which is a function from natural numbers, the table, tab, might be a list of values of the function, in order. Then $fib\ n = tab\ !!\ n$. The table can be constructed by mapping an implementation of fib over the domain of the function, [0..]. That implementation consists of the kernel of the recursion, but with recursive calls implemented by table lookup.

```
> tabulate :: ((Int -> a) -> (Int -> a)) -> (Int -> a)
> tabulate kernel = fun
>                   where fun = (tab !!)
>                         tab = map (kernel fun) [0..]
```

Think of *tabulate* as a more efficient implementation of fix, specialised to the type $Int \to a$, with arguments restricted to non-negative values of Int.

$$
\begin{array}{ll}
 & fun\ n \\
= & \{\text{ local definition of } fun \} \\
 & tab\ !!\ n \\
= & \{\text{ local definition of } tab \} \\
 & (map\ (kernel\ fun)\ [0..])\ !!\ n \\
= & \{ (map\ f\ xs)\ !!\ n = f\ (xs\ !!\ n), \text{ for infinite } xs, \text{ by induction on } n \} \\
 & kernel\ fun\ n
\end{array}
$$

so $fun = kernel\ fun$ and (because it is unique) this is $fix\ kernel = fib$.

With this definition, $tabulate\ fib_K = fib$ is an implementation of fib which only ever calculates $fib_K\ (\ldots)\ n$ once for each value of n.

The table tab is infinite, but only a finite amount of it (the first $n+1$ elements) will be explored and filled in for any call of $tabulate\ fib_K\ n$. However, that piece of the table which is instantiated grows with n so the cost of reducing the time taken by the program is an increase in the space which it must occupy.

(This is not a linear implementation of fib, because it calls fun about twice for each natural number less than n, and fun is linear in its numerical argument, so the resulting implementation of fib is quadratic.)

Strictly speaking, the name *dynamic programming* is usually used to mean tabulation, with predetermined bounds on the table to be used, along with a schedule for evaluating the elements of the table. The schedule guarantees that each entry in the table is filled in before it is looked up. In our program, we rely

on lazy evaluation to evaluate the table entries in an order which provides the answers when they are needed. This works provided that there is an ordering on the arguments with respect to which recursive calls are smaller than the call which causes them. Such an ordering must have existed in order for the original recursive function to terminate. In the case of *fib*, this is just the usual ordering on numbers.

14.2 Tabulating results

The calculation of solutions to a countdown problem can reorganised to use a function which lists the solution expressions that use all of the numbers in some list. (The reasons for wanting to do that will become clear later.)

```
> solutions :: ([Int] -> [Result]) -> [Int] -> Int -> [Expr]
> solutions results ns n = [ e | (e,v) <- results ns, v == n ]

> results :: [Int] -> [Result]
> results ns = [ ev | ys <- subs ns, ev <- fix resultsK ys ]
```

The *results* function lists those expressions that use exactly *ys* for each subsequence *ys* of its argument *ns*, using a recursion described by $results_K$.

```
> resultsK :: ([Int]-> [Result]) -> ([Int] -> [Result])
> resultsK f [n] = [ (Val n, n) | n > 0 ]
> resultsK f ns  = [ (App o l r, apply o x y)
>                  | (ls, rs) <- subsplits ns,
>                    ((l,x),(r,y)) <- f ls `cp` f rs,
>                    o <- ops,
>                    needed o l r,
>                    useful o x y ]
```

This recursion filters for *useful* arrangements of values of subexpressions, but also for the associativity properties eliminated by *gen* from the previous lecture.

Binary expressions are constructed by dividing *ns* into two subsequences, rather than just an initial segment and a final segment. The *subsplits* function divides a non-empty list into a pair of non-empty subsequences in all possible ways. (They must be non-empty sequences, because the program later relies on *the other* element of the pair being smaller than the list being split.)

```
> subsplits :: [a] -> [([a], [a])]
> subsplits (x:[]) = []
> subsplits (x:xs) = [ ([x],xs), (xs,[x]) ] ++
>                    concat [ [ (x:ls,rs), (ls,x:rs) ]
>                           | (ls,rs) <- subsplits xs ]
```

This leads to fewer arguments to *results* than the strategy in the previous lecture, because the numbers in the argument of $results_K$ are always in the order in which they appeared in the problem.

There is as yet no tabulation in this new implementation, but it is slightly faster than the fastest solution from the previous lecture.

```
Lecture14> head (Lecture13.approx Lecture13.usefulresults numbers 831)
(7+(3*((1+10)*25)),832)
(1.89 secs, 849,541,120 bytes)

Lecture14> head (approx results numbers 831)
(7+(3*((1+10)*25)),832)
(0.94 secs, 416,264,712 bytes)
```

14.3 Naïve tabulation

A table for the *results* function has to represent a function $[Int] \to Int$, with domain $subs\ ns$ for some six numbers ns.

One perfectly general solution represents an $Eq\ a \Rightarrow a \to b$ by an *association list* of pairs $[(a, b)]$, which for clarity later, we tag with the constructor *Mapping*

```
> data Mapping a b = Mapping [(a,b)]
```

then the association list with domain xs is just a list of pairs of elements of the domain together with their image under the function which the list represents.

```
> toMapping :: [a] -> (a -> b) -> Mapping a b
> toMapping xs f = Mapping [ (x, f x) | x <- xs ]
```

Values of $f\ x$ can be recovered from the map by finding the (or the first) value paired with x.

```
> getMapping :: Eq a => Mapping a b -> a -> b
> Mapping m `getMapping` x = head [ b | (a,b) <- m, a == x ]
```

With this implementation of a mapping the fixed point of $results_K$ can be tabulated with an association list indexed by the subsequences of ns.

```
> tabresults :: [Int] -> [Result]
> tabresults ns = [ ev | xs <- dom, ev <- fun xs ]
>                 where fun = (tab `getMapping`)
>                       tab = dom `toMapping` (resultsK fun)
>                       dom = subs ns
```

This function constructs the table once only for each problem (one with numbers ns) so calls $results_K$ only $2^6 = 64$ times.

```
Lecture14> head (approx tabresults numbers 831)
(7+(3*((1+10)*25)),832)
(0.33 secs, 143,709,704 bytes)
```

However it makes many calls of *getMapping*, each of which involves exploring the table, which is itself 64 long.

14.4 Using a Trie for tabulation

The domain of the mapping is a set of lists, and a possibly more efficient strategy for tabulating a prefix-closed set of lists is to use a *Trie* (possibly meant to be pronounced *tree*) which is a particular form of *rose tree*.

```
> data Trie a b = Trie b (Mapping a (Trie a b))
```

just as every rose tree contains a forest of subtrees, you can think of this trie as containing a (small) forest of tries

```
> data Trie a b = Trie b (Copse a b)
> type Copse a b = Mapping a (Trie a b))
```

The nodes of a *Trie a b* for some function $f :: [a] \to b$ correspond to a set of prefixes of some lists from $[a]$ together with the values of the function at those prefixes. The root contains $f\ [\,]$; the copse contains the values of f at non-empty lists with $f(x : xs)$ in the trie found by looking for x in the mapping in the root:

```
> getTrie :: Eq a => Trie a b -> [a] -> b
> (Trie y _) `getTrie` [] = y
> (Trie _ m) `getTrie` xs = m `getCopse` xs

> getCopse :: Eq a => Copse a b -> [a] -> b
> m `getCopse` (x:xs) = (m `getMapping` x) `getTrie` xs
```

How to construct a trie for a function on the subsequences of a list? One way is to consider the subsequences generated in lexicographical order of the positions in the original list.

```
> subs :: [a] -> [[a]]
> subs xs = [ y:zs | (y:ys) <- tails xs, zs <- [] : subs ys ]
```

Here the *tails* are the non-empty tails of a list.

```
> tails :: [a] -> [[a]]
> tails [] = []
> tails xs = xs : tails (tail xs)
```

An analogous recursion produces a trie for the subsequences of xs.

```
> toCopse :: [a] -> ([a] -> b) -> Copse a b
> xs `toCopse` f = Mapping [ (y, ys `toTrie` (f.(y:)))
>                          |  y:ys <- tails xs ]

> toTrie :: [a] -> ([a] -> b) -> Trie a b
> xs `toTrie` f = Trie (f []) (xs `toCopse` f)
```

Here the function $f \cdot (y :)$ is tabulated by the trie that is reached from the root by looking up y, so a value found by looking up a path ys in that subtrie can be found by looking up $y : ys$ from the root.

The tabulation of the results looks very similar to the earlier one, except that *toTrie* produces a trie with a domain that is not its list argument, but the subsequences of that argument

```
> trieresults :: [Int] -> [Result]
> trieresults ns = [ ev | xs <- subs ns, ev <- fun xs ]
>                  where fun = (tab `getTrie`)
>                        tab = ns `toTrie` (resultsK fun)
```

and since we only want to tabulate values for non-empty sequences

```
> copseresults :: [Int] -> [Result]
> copseresults ns = [ ev | xs <- subs ns, ev <- fun xs ]
>                   where fun = (tab `getCopse`)
>                         tab = ns `toCopse` (resultsK fun)
```

and the resulting program is significantly faster. The effect is more noticeable with bigger problems, for example with seven numbers a trie is a clear factor of two better than an association list.

14.5 Approximate solutions

Countdown rewards the nearest solution to a problem, at least provided that it is within ten of the target. The nearest solution might be calculated by

```
> approx :: ([Int] -> [Result]) -> [Int] -> Int -> [Result]
> approx results ns n
>  = sortOn (near n) [ r | cs <- choices ns, r <- results cs]
>    where near n (_,v) = abs (n-v)
```

The library function

$$Data.List.sortOn :: Ord\ b \Rightarrow (a \to b) \to [a] \to [a]$$

sorts a list so that $map\ f\ (sortOn\ f\ xs)$ is in ascending order.