The Generation of Strings from a CFG using a Functional Language

Bruce McKenzie
University of Canterbury
Christchurch, New Zealand
B.McKenzie@cosc.canterbury.ac.nz

September 1, 1997

Abstract

It is common to describe the input to many programs and systems in terms of a context free grammar (CFG). In order to test programs that process strings generated from such grammars it would be extremely useful to have effective methods of generating strings from the grammar itself.

This paper explores a number of interesting issues that arise in generating such strings in the context of functional programming languages. A number of features commonly provided in functional languages, such as lazy evaluation and infinite data structures, along with the ease with which memoization of function can be implemented are surprisingly useful in the solution of this problem.

The paper presents two distinct solutions to this problem. The first presents a method that generates all possible strings in order of increasing length. The second shows how to generate strings from the grammar at random such that all strings of length n are produced with equal probability.

It is often necessary in practice to perform manipulations by hand on a CFG in order to convert it into some particular form. For example to use recursive descent parsing techniques it is necessary to remove left recursion. A key requirement of the second random generation approach are counts of the number of strings of length $m \leq n$ generated by the grammar. We show how these counts can be useful in providing a partial check that such manipulations of a grammar have been performed without (accidentally) changing the language generated.

1 Introduction

There are many situations where the input to a program is described by a context free grammar (CFG). In such programs it is common to construct a parser to recognise strings generated by the grammar either by hand, using techniques such as recursive descent [1], or using a tool such as Bison [2]. Normally strings are then generated by either human users or possibly by other programs.

However in order to test such parsers we require sample strings from the language generated by the grammar. Although it may be possible to collect such strings from sources such as user input, it is often the case that these may not exercise all the productions of the grammar and may not be available in a wide range of string lengths. It would be useful if such sample strings were generated automatically from the grammar itself.

Furthermore if we wish to test the error repair properties of the parse algorithm it is necessary to have a source of strings with errors. For such tests we do not require strings that are completely unrelated to those generated by the grammar but are at least approximately correct. By generating correct strings and introducing errors by deleting, inserting and changing tokens we have a controlled source of strings for testing purposed. Testing the properties of a least-cost error repair algorithm in the context of shift-reduce parsers [3] was our initial motivation for this problem.

A number of the language features commonly available in functional programming languages — lazy evaluation, infinite data structures, along with the ease with which memoization of functions can be implemented — are extremely useful in the solution of this problem.

We begin in Section 2 on the problem of producing all strings generated by the grammar and present a Haskell program to solve this problem. In Section 3 we instead concentrate on selecting a sub-set of strings generated by the grammar by choosing strings at random from the grammar. Again a Haskell program is presented to solve this problem and its computation complexity is studied. Finally in Section 4 we present some conclusions and suggestions for future work.

2 Generating all strings from a CFG

For this section the important properties of functional languages we will use are those of lazy evaluation and the ability to deal with (potentially) infinite data structures. Our basic model will be to allow each terminal and non-terminal in the grammar to be represented by a data structure consisting of a (possibly infinite) list of strings that can be generated from the symbol. For example the grammar ntA $\rightarrow$ `a` | `b` ntA `c` would result in the lists:

```
> a,b,c,ntA :: [String]
> a         = ["a"]
> b         = ["b"]
> c         = ["c"]
> ntA       = ["a", "bac", "bbacc", ....]
```

The lists will be held in order of increasing length measured either by the number of terminal symbols they contain or the total number of characters as appropriate. The programs we shall present shall use the number of terminals but will be parameterised by a function. This could be easily replaced by one to order them differently.

We will then be able to ask for strings from the list associated with the start symbol. For example we could print the first 100 such strings:

```
> main :: IO()
> main = mapM_ print (take 100 ntS)
```

By defining Haskell operators that correspond to the grammar operators of alternative (|) and concatenation (juxtaposition) we allow the Haskell program to closely mirror the grammar. As these operators are already used for other purposes in Haskell we shall use ||| and +++ instead.

```
> infixl 5 |||
> infixl 6 +++                          -- tighter than |||

> ntA = a ||| b +++ ntA +++ c              -- A → a | b A c
```

The operator ||| is straightforward to define as it simply involves a merge of two sorted lists.

```
> merge2 :: (a → a → Ordering) → [a] → [a] → [a]
> merge2 cmp as [ ]          = as
> merge2 cmp [ ] bs          = bs
> merge2 cmp (a:as) (b:bs) = case cmp a b of
>                                GT → b: (merge2 cmp (a:as) bs)
>                                _   → a: (merge2 cmp as (b:bs))

> (|||) :: [a] → [a] → [a]
> a ||| b = merge2 cmpSymLen a b
```

The comparison function enables the order in which the strings are generated to be controlled. We will assume that the strings are represented by keeping the generated strings as lists of tokens and so the following results in strings with increasing numbers of tokens.

```
> cmpSymLen      :: [a] → [a] → Ordering
> cmpSymLen [ ] _ = LT
> cmpSymLen _ [ ] = GT
> cmpSymLen (_:as) (_:bs) = cmpSymLen as bs
```

Note that only the start of the longer string is needed in order to determine the appropriate order.

The operator +++ is more complex as it requires the cross product of two potentially infinite sorted lists producing a sorted list as output. For example given the productions:

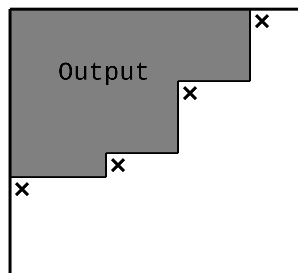

Figure 1: Tableau showing strings already output and the 'corners' from among which the next shortest string will be selected.

$$\begin{array}{lcl} \text{ntS} & \rightarrow & \text{ntA ntB} \\ \text{ntA} & \rightarrow & \mathtt{a} \mid \mathtt{a}\ \text{ntA} \\ \text{ntB} & \rightarrow & \mathtt{b} \mid \mathtt{b}\ \mathtt{b}\ \text{ntB} \end{array}$$

we need to combine:

```
> ntA = [ "a", "aa", "aaa", ... ]
> ntB = [ "b", "bbb", "bbbbb", ... ]

> ntS = ntA +++ ntB
>     = [ "a", "aa", "aaa", ... ] +++ [ "b", "bbb", "bbbbb", ... ]
>     = [ "ab", "aab", "aaab", "abbb",  ... ]
```

Given two lists $a_1, a_2, \ldots$ and $b_1, b_2, \ldots$ there is no question that the front of the cross product list will be the concatenation of the strings from the front of each list $a_1.b_1$, but is the second element $a_1.b_2$ or $b_1.a_2$? To decide we must compare the two strings and select the 'shortest'. Then we must decide on the next shortest and so on.

To keep track of which strings need to be compared to determine the next shortest, note that at any time we will have a region of strings that have been output and then a series of 'corners' at the frontier of the region from among which the next shortest string will be located. This is illustrated in Figure 1 where the critical 'corners' are shown as crosses. Note that every string to the right and below any corner cannot be shorter than that at the corner. All that is required is to determine where these corners are, and when we have chosen the corner with the next shortest string, decide how to update the list of corners. As can be seen from Figure 2, when a corner is selected it can result in either zero, one, or two new 'corners'.

We will maintain the corners of the frontier as a list kept in order of increasing string length. Each corner will store the string along with the row and column index values.

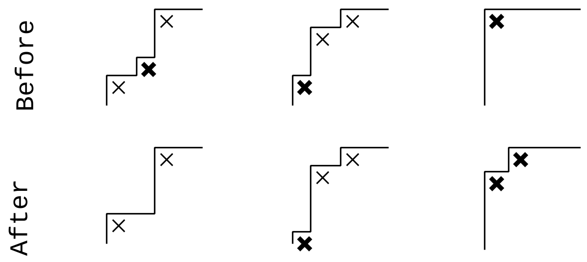

Figure 2: Choosing a string from a corner can result in either zero, one or two new corners.

```
> type Corner = (Symbols,Int,Int)
> type Frontier = [Corner]

>          cmpVal                 :: Corner → Corner → Ordering
>          cmpVal (v1,_,_) (v2,_,_) = cmpSym v1 v2
```

We can now define our Haskell operator to concatenate two ordered lists of strings into an ordered list.

```
> ( +++ ) :: [Symbols] → [Symbols] → [Symbols]
> a +++ b = prod cmpSymLen a b

> prod :: (Symbols → Symbols → Ordering) →
>         [Symbols] → [Symbols] → [Symbols]
```

The initial frontier consists of the single corner at row 0, col 0 corresponding to the concatenation of the first string from either list.

```
> prod cmpSym l1 l2 = getfrontier initfrontier
>    where initfrontier :: Frontier
>          initfrontier = [(initval,0,0)]
>          initval = (head l1) ++ (head l2)
```

After the front (shortest) element is removed from the frontier then there are two potential new entries to be added, those at $(r+1,c)$ and $(r,c+1)$ where (r,c) is the row and column number of the front element.

```
>         getfrontier          :: Frontier → [Symbols]
>         getfrontier []        = []
>         getfrontier frontier  = a: getfrontier frontier'''
>            where ((a,r,c) :frontier') = frontier
>                     frontier''        = addrow frontier'  (r+1) c
>                     frontier'''       = addcol frontier'' r     (c+1)
```

We need only add a corner to the frontier if there is none already present with the same column (row) when we add a row (column) as otherwise we would know it was not shorter than the one already there.

```
>         addrow        :: Frontier → Int → Int → Frontier
>         addrow fs r c = if r `elem` rs then fs
>                         else insert r c fs
>               where rs = map (\(_,r,_)→r) fs
>           addcol         :: Frontier → Int → Int → Frontier
>           addcol fs r c = if c `elem` cs then fs
>                              else insert r c fs
>                 where cs = map (\(_,_,c)→c) fs
```
The function insert maintains the frontier sorted by length but must be careful if either list is finite:
```
>           insert              :: Int → Int → Frontier → Frontier
>           insert r c frontier =
>                case nth r l1 of
>                Nothing → frontier
>                (Just a) → case nth c l2 of
>                              Nothing → frontier
>                              (Just b) → insertBy cmpVal (a++b,r,c) frontier
```
where nth acts much like the operator (!!) for lists but detects attempts to select an element off the end of the list.
```
> nth                  :: Int → [a] → Maybe a
> nth 0 (x:_)          = Just x
> nth n (_:xs) | n > 0 = nth (n−1) xs
> nth _ []
```
It is straightforward to use the alternation and concatenation operators to add extra extended BNF featurs such as: the Kleene closure operators, of both the zero or more a^* and one or more variety a^+, and optional $a?$ operator.

Finally we can take a grammar and edit it adding these operators as in the following initial productions for a Pascal grammar.
```
> program        = tPROGRAM +++ newident +++
>                    external_files +++ tSEMI +++ block +++ tDOT

> external_files
>                = opt ( tLPAREN +++ newident_list +++ tRPAREN )

> block          = (star declaration) +++ statement_part

> declaration    = label_dcl_part ||| const_dcl_part
>                                 ||| type_dcl_part
>                                 ||| var_dcl_part
>                                 ||| proc_dcl_part
```
Printing out the first few strings generated gives:
```
? mapM_ print (take 5 (map showstr program))
"program i ; begin end ."
"program i ; begin ; end ."
"program i ; begin i end ."
"program i ; begin i ; end ."
"program i ; begin 0 :  end ."
```

2.1 Limitations of the method

It is well known that many parsing methods have problems with certain grammar features such as left-recursion in recursive descent or LL parsing. Left recursion is equally a problem for the algorithm of the previous section and furthermore there are also subtle problems with the interaction between lazy evaluation and maintaining the frontier in sorted order. For example although there are no problems with the grammar fragment x = a ||| a +++x, reversing the two alternatives as in x = a +++x ||| a results in infinite recursion. As can be seen, our definition of the operator ||| is not symmetric but favours the left alternative when the strings are equal in length. However if there is sufficient information available to determine which is the longer string without recursing, as in the example x = a +++a +++x ||| a, then the generation proceeds without problems.

These problem fragments can be overcome by rewriting the grammar much as is done when using LL parsing and using the Kleene closure operators to eliminate troublesome Haskell functional recursion. An alternative approach is to hold each list of strings as an infinite list of lists where the first element is the list of strings of length 0, the next of length 1 and so on. Concatenation now requires no comparison operations to be performed but is essentially the diagonalization operation such as that provided by Miranda[1] and does not result in infinite recursion. However an unfortunate consequence of this approach is that **all** lists, even those for the terminal symbols, must be held as infinite lists or the problem reappears when attempts are made to test if there is a next element in the list (of lists). Another potential solution may be afforded by the mix operator described in the monadic combinator parsing approaches of [4] and [5].

However as Table 1 indicates, generating all strings for a grammar is not a particularly practical approach even with quite short lengths for any but very simple grammars. This table gives the number of strings containing n terminals for three well-known languages. Clearly the exponential growth of these counts will mean only very short strings can be generated in practice before time and space constraints become excessive.

If we wish to generate longer strings then it is only practical to generate a subset of all strings and do this by selecting these at random in some manner. The next section investigates this in detail.

3 Generating strings at random

It would initially seem that generating strings at random would be easily achieved by the following obvious algorithm:

1. Start with the start symbol (S),
2. choose one of its productions at random (eg. S $\rightarrow$ A b C),
3. repeat recursively with A and then with C.

However this approach has a number of problems.

[1]Miranda is a trade mark of Research Software Limited

n	Pascal	Modula-2	Java
1	0	0	1
2	0	0	1
3	0	0	3
4	0	0	5
5	0	0	18
6	1	1	61
7	2	6	253
8	6	23	1441
9	37	91	11685
10	194	371	119173
11	2351	2037	1.49×10^6
12	18542	12563	2.45×10^7
13	197136	108160	5.86×10^8
14	1.65×10^6	907089	1.81×10^{10}

Table 1: The number of strings of n terminals for three well-known programming languages.

Firstly observe that it may not terminate. For example the grammar with productions $S \rightarrow SS|a$ terminates with a probability of $\frac{1}{2}+\frac{1}{8}+\frac{2}{32}+\frac{5}{128}+\frac{14}{512}+\cdots \approx 0.873595$. Furthermore it is not fair in that not all strings of the same length are generated with equal probability. For example the grammar $S \rightarrow a|A; A \rightarrow b|c$, will generate an a with probability $\frac{1}{2}$ but a b only with probability $\frac{1}{4}$. Finally it is grammar dependent in that different grammars generating the same language can produce the same string with different probabilities.

A far better approach would be to treat all strings of length n by the grammar equally so that each string is equally likely to be generated. As explained in [6] this can be achieved by calculating the number of strings of length n generated by each production and each postfix of any right hand side of a production.

The full details of the algorithm are available in [6] but the idea of the approach can easily be illustrated by considering a small example. Given the following (tree) grammar

$$\begin{array}{lcl} \mathrm{F} & \rightarrow & \mathrm{d\ T\ u} \mid \mathrm{d\ T\ u\ F} \\ \mathrm{T} & \rightarrow & \mathrm{x} \mid \mathrm{x\ F} \end{array}$$

consider the generation of a string of length 12 from F.

To decide whether to choose $F \rightarrow dTu$ or $F \rightarrow dTuF$ we note that the first production produces 5 strings of length 12 while the second produces 9. So when selecting which initial production to generate we choose the first production with probability $\frac{5}{14}$ and the second with probability $\frac{9}{14}$.

Now consider the case when the second production, $F \to dTuF$, was selected. We generate a "d" and are left with the remainder of the production TuF to produce the further 11 symbols. This requires us to decide how many of these 11 symbols should come from T and how many from the remainder. I.e. if the symbol T generates l symbols then the remaining (11-l) must come from uF.

To decide on the value l fairly we need to know, for each possible $l = 1, 2, \ldots 10$, how many strings will be generated. If l is 1 then there are 5 strings, there are 2 strings when l is 4 or 7 and no strings possible for all other values of l. Hence we choose l to be 1 with probability $\frac{5}{9}$ and l to be 4 or 7 with probability $\frac{2}{9}$. Having decided how many strings the symbol T must generate we use this to decide on which production to use and repeat this process recursively.

So we observe that if we can calculate the counts then we can generate random strings fairly.

Given productions:

$$P = \{\pi_{ij} : N_i \to \alpha_{ij} | i = 1 \ldots r, j = 1 \ldots s_i\}$$

where:

$$\alpha_{ij} = x_{ij1}x_{ij2} \ldots x_{ijt_{ij}}$$

then [6] derives functions that generate the required counts. Given a non-terminal N_i then function $f_i(n)$ allows us to decide how to weight each production, as it gives the number of strings of length n generated by each possible production with left hand side N_i.

$$f_i(n) = [\ \|\alpha_{ij}\|_n \mid j \leftarrow 1 \ldots s_j]$$

where $i = 1 \ldots r$.

Having decided on a production $N_i \to \alpha_{ij}$, then the function $f'_{ijk}(n)$ gives the number of strings of length n generated by the final symbols $x_{ijk} \ldots x_{ijt_{ij}}$ from the right hand side of the production π_{ij} for each of the possible ways in which the n symbols can be split between the first symbol x_{ijk} and the remaining symbols.

$$f'_{ijk}(n) = [\ \|x_{ijk}\|_l \times \|x_{ij(k+1)} \ldots x_{ijt_{ij}}\|_{n-l} \mid l \leftarrow 1 \ldots n - t_{ij} + k]$$

where $i = 1 \ldots r, j = 1 \ldots s_j, k = 1 \ldots t_{ij}$.

The length of the list is $n - t_{ij} + k$ and not n because we disallow productions of the form $A \to \epsilon$ so no symbol can generate the empty string. This means all strings generated by the symbols $x_{ij(k+1)} \ldots x_{ijt_{ij}}$ will have length of at least $t_{ij} - k$. This restriction is necessary to ensure forward progress is made.

It should be noted that if the grammar is ambiguous then these functions will over-represent any ambiguous string by the number of distinct derivation trees in the grammar for that string.

These functions can be converted to mutually recursive functions in Haskell using the optimisation that terminal symbols always produce only a single string.

However as they stand the functions f and f' are **very** expensive to evaluate and have a time complexity exponential in the length n.

One possible way to improve this exponential behaviour is to use dynamic programming techniques by first calculating and storing the n=1 entries and then using these in calculating the n=2 entries and so on. There are two disadvantages of this approach: firstly we would calculate many entries we never needed to consult when generating random strings and secondly with chain rules of the form $N_i \to Nj$ care would need to be taken to ensure f_i was evaluated before f_i.

A better solution is obtained by using functional language facilities. Lazy evaluation avoids calculating counts that are not required. Experiments with Pascal and Modula-2 suggest that this results in a saving of approximately 30% compared to the number of values that would be calculated using dynamic programming. This can be combined with memoization[7] to reduce the cost of repeated calls with the same arguments. Memoizing a function means when we call f 5 we evaluate it as normal but then save the result so if f 5 is ever needed again we can return the value immediately. The resulting space overhead depends on the number of different possible values of the parameter. This is straightforward to implement in Haskell using arrays:

```
> memoize :: (Ix a) ⇒ (a,a) → (a → b) → a → b
> memoize bds f = (!) arr
>     where arr = array bds [ (t, f t) | t ← range bds ]
```

For example we can produce a memoized version of the Fibonacci function:

```
> fib 0 = 1
> fib 1 = 1
> fib n = fib (n−1) + fib (n−2)

> fastfib = memoize (0,25) fib′
>   where
>       fib′ 0 = 1
>       fib′ 1 = 1
>       fib′ n = fastfib (n−1) + fastfib (n−2)
```

which reduces its exponential complexity to linear complexity.

The implementation of this in Haskell is simplified by storing the grammar in array structures with each array being only as long as required:

```
> type IdxSymbol  = Int -- a symbol
> type IdxSymbols = Array IdxSymbol String -- each symbols representation

> type IdxRhs      = Array Int IdxSymbol -- RHS of a single production
> type IdxRules    = Array Int IdxRhs -- RHSs with the same LHS non−terminal
> type IdxGrammar = Array IdxSymbol IdxRules -- all productions
```

As a result the parameters to the f and f′ functions are then directly related to the indices of these arrays.

Both functions can be memoized at each level of the array indexing and a direct translation of the original functions gives:

```
> type Length = FloatOrDouble -- numeric type for counts
> type NTLen = IdxSymbol→Int              → [Length]  -- type of f   function
> type RhsLen = IdxSymbol→Int→Int→Int → [Length]  -- type if f′ function
> lens :: IdxGrammar → Int → (NTLen,RhsLen)
> lens rules n = (f,f′)
>   where

>       f  :: IdxSymbol → Int → [Length]
>       f = memoize (bounds rules) fA
>         where
>             fA a  = memoize (1,n) fAk
>               where
>                   fAk k = [sum (f′ a i 1 k) | i ← indices (rules!a)]
>      f′ :: IdxSymbol → Int → Int → Int → [Length]
>      f′ = memoize (bounds rules) f′A
>        where
>            f′A a  = memoize (1,am) f′Ai
>              where
>                  am = snd (bounds (rules!a))
>                  f′Ai :: Int → Int → Int → [Length]
>                  f′Ai i = memoize (1,aip) f′Aij
>                    where
>                        aip = snd (bounds (rules!a!i))
>                        f′Aij :: Int → Int → [Length]
>                        f′Aij j = memoize (0,n) f′Aijk
>                          where
>                              xaij = rules!a!i!j
>                              f′Aijk :: Int → [Length]
>                              f′Aijk 0 = [ ]
>                              f′Aijk k =
>                                  if isterm xaij then
>                                     if j == aip then
>                                        if (k==1) then [1.0] else [0.0]
>                                     else
>                                        [sum (f′ a i (j+1) (k−1))]
>                                  else
>                                     if j == aip then
>                                        [sum (f xaij k)]
>                                  else
>                                        [ sum(f xaij l) ‘lazyprod‘
>                                          sum(f′ a i (j+1) (k−l)) |
>                                                l ← [1..k−aip+j]]
```

A final optimisation that is very effective is to make the multiplication operator lazy as well.

```
>                         0 ‘lazyprod‘ _ = 0
>                         x ‘lazyprod‘ y = x∗y
```

Given the above functions, generating strings requires only a stream of random values. These enable choices to be made using the probabilities obtained by calls to the f and f′ functions.

3.1 Algorithm Complexity

Before memoization, the time complexity was exponential and the space $\mathcal{O}(1)$ although there are significant space overheads associated with the extreme stack recursion. After memoization, the time complexity is reduced to $\mathcal{O}(n^2)$ while the space has risen to $\mathcal{O}(n^2)$ to allow saving of results. However many of the recursive overheads will have been reduced which will offset some of this. The string generation process is $\mathcal{O}(n)$ in both time and space.

4 Conclusions and future work

It should be observed that the string counts can be useful in providing an approximate but quite effective test that two grammars generate the same language. For example one might have been derived from the other by hand manipulation. If the number of strings of length 1,2,... generated by the two start symbols are in agreement then we can have greater confidence that the manipulation was performed correctly. This test can be further strengthened by comparing the counts up to the point when every production has been used; as evidenced by each $f_{ij1}(n)$ count having generated a non-zero value.

The form of the recursive equations defining f and f' are remarkably similar to the convolution functions [8] for which the Fast Fourier Transform algorithm reduces an $\mathcal{O}(n^2)$ algorithm to $\mathcal{O}(n \log n)$ time. Although there appears to be significant impediments to using such techniques owing to the severe mutual recursion between f and f', it would be interesting to see if the formulation of FFT in functional languages[9] and lazy evaluation might be combined in some manner.

It is clear that a number of features of lazy functional programming languages have combined in a significant manner to simplify this problem. Although the memoization could be performed in conventional languages it is not nearly so easy and the duplication of lazy evaluation would be complex to code.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] R. Corbett and R. Stallman. *Bison: Gnu parser generator.* Free Software Foundation, Cambridge, Mass., 1991. Tex info documentation.

[3] Bruce J. McKenzie, Corey Yeatman, and Lorraine de Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4):672–689, July 1995.

[4] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[5] Paul Lickman, Oege de Moor, and Philip Wadler. Fixing combinators parsers. In preparation, 1996.

[6] Bruce McKenzie. Generating strings at random from a context free grammar. Technical report TR-COSC 10/97, Department of Computer Science, University of Canterbury, 1997.

[7] D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[8] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[9] Geraint Jones. Deriving the Fast Fourier algorithm by calculation. In Kei Davis and John Hughes, editors, *Glasgow Functional Programming Group Workshop*, Springer Workshops in Computing. Springer, 1990. presented at the Glasgow Functional Programming Group Workshop, Fraserburgh, Scotland, 21-23 August 1989.