A Functional Program For The Fast Fourier Transform

Fer-Jan de Vries

Department of Philosophy
University of Utrecht

Logic Group
Preprint Series
No.19

Department of Philosophy

University of Utrecht

A Functional Program For The Fast Fourier Transform

Fer-Jan de Vries

March 1987

Department of Philosophy
University of Utrecht
Heidelberglaan 2
3508 TC Utrecht
The Netherlands

* the author is partially sponsored by the Dutch Ministry of Science and Education through the project "Parallel Reduction Machine".

Contents

1. Introduction

This paper is written as a contribution to the Parallel Reduction Machine Project. Its purpose is to present a *functional* program for a well-known application of the fundamental algorithmic method *Fast Fourier Transform* for multiplication of polynomials. This in order to verify experimentally two claims by functional programmers [BvL]:

(i) functional programming is good for writing structured software; better so than the so-called *imperative von Neumann-languages*.

(ii) functional programming allows for a parallel evaluation of subexpressions, provided a proper implementation.

With pleasure I acknowledge discussions with Henk Barendregt, Jan Bergstra, Karst Koymans, Marc van Leeuwen, Gerard Renardel, Piet Rodenburg and Albert Visser.

2. Miranda

We take the functional programming language *Miranda* (meaning: to be admired) as our representative of a modern functional programming language for several reasons:

(i) Miranda has a pleasantly working implementation (we used version 0.378);

(ii) it has a type treatment more or less similar to the language TALE proposed by some workers on the Parallel Reduction Machine Project.

Information on Miranda can be found in the on-line Miranda System-manual, and in articles by the creator of Miranda, D.A. Turner [T84, T85]. Roughly speaking one can say that -like other modern functional programming languages- Miranda is based on the idea of using recursion equations to define data types and functions between them [T84]. The compiler of Miranda is able to deduce -if not specifically declared- the types of functions.

Reading a script in Miranda is mostly self-explaining. We give a short overview on types and some notations. Text in Miranda will be in **bold face**.

2.1 Types in Miranda

Primitive types are **num**, **bool** and **char**. Furthermore there are *generic types* indicated by ***, **, ***** ... etc. From types one can construct other types: *function types* $T_1 \rightarrow T_2$, *list types* **[T]** and *tuple types* $(T_1, ..., T_n)$.

The type system of Miranda allows user defined types: *type synonyms, algebraic types* and *abstract data ypes.*

(i) Type synonyms are simple: one gives new names for already constructed types. Example: **plane == (num,num).**

(ii) Algebraic types can be best explained by examples.

Example 1: **tree::= Niltree | Node num tree tree**. (Cf. [T85].) **Niltree** and **Node** are so-called type constructors. One can think of **tree** as a kind of free algebra generated by numbers and the two constructors, however infinite words are allowed: the infinite object **bigtree = Node 1 bigtree bigtree** is an element of **tree.**

Example 2:
```
ord_pairs ::= Pair num num
Pair a b => Pair b a, a>b.
```

ord_pairs is constructed from numerals and the type constructor **Pair** which is subject to an associated law. **Pair** switches the order of **a** and **b** whenever **a** is larger than **b**. In contrast to the other example Turner would call this an unfree algebra.

(iii) Abstract data types are nothing but abstract signatures, that will be made concrete somewhere further on in the script via so-called implementation equations.

Example:
```
abstype complex_numbers
with add, subtract, multiply :: complex_numbers->complex_numbers
```

Now one can use the functions **add** and **subtract** and **multiply** in the script. And somewhere else in the script one tells which concrete type represents the **complex_ numbers**:

```
complex_numbers == (num, num)
add (a,b) (c,d) = (a+c,b+d).
```

2.1.1 Example: the type of polynomials in Miranda

For a mathematician it is convenient to think of polynomials as sequences of numbers. E.g. $a_0+a_1x+...+a_nx^n$ is represented by $[a_0,...,a_n]$. This choice -instead of $[a_{N-1}, ..., a_1, a_0]$- facilitates the definition of polynomial addition. Then, as in daily practice, she steps lightly over the fact that equality of polynomials differs from equality of sequences.

One solution in Miranda for this approach is to construct a concrete type **[num]** (**num** is just the type of real numbers in Miranda) containing sequences of reals together with a definition of equality for this type:

$\textbf{poly}_1$ :: [num]

```
pol_eq1 :: poly1→poly1→bool
pol_eq1 f g =   rev_pol_eq1 reverse f reverse g
                where
                rev_pol_eq1 [] [] = True
                rev_pol_eq1 (a:f) [] =   a=0 & rev_pol_eq1 f []
                rev_pol_eq1 [] (b:g) =   b=0 & rev_pol_eq1 [] g
                rev_pol_eq1 (a:f) (b:g) =   a=b & rev_pol_eq1 f g
```

(**reverse** just reverses the order in a list)

For this type polynomial addition can be defined as follows:

```
add1 :: poly1→poly1→poly1
add1 f [] = f,
add1 [] g = g,
add1 (a:f) (b:g) = a+b:add1 f g
```

Note. This definition means that Miranda will see no difference between a sequence of numbers and a sequence of numbers that represents a polynomial. If the programmer wants to insist on such a intensional difference, then there is a way out for him.

The notion of abstract data type in Miranda creates such a formal difference:

```
abstype poly2
with add2 :: poly2→poly2→poly2
     pol_eq2 :: poly2→poly2→bool

poly2 == [num]

pol_eq2 f g =   rev_pol_eq2 reverse f reverse g
                where
                rev_pol_eq2 [] [] = True
                rev_pol_eq2 (a:f) [] =   a=0 & rev_pol_eq2 f []
                rev_pol_eq2 [] (b:g) =   b=0 & rev_pol_eq2 [] g
                rev_pol_eq2 (a:f) (b:g) =   a=b & rev_pol_eq2 f g
add2 f [] = f,
add2 [] g = g,
add2 (a:f) (b:g) = a+b:add2 f g
```

2.2 Short overview of list notations in Miranda

Basically, lists are written with square brackets and comma's: **[0,3,2,4,4]** or **[]**.

(i) # **x** is the length of the list **x**

(ii) **x!n** is the n-th element of list **x** (nasty detail: subscripting starts at 1)

(iii) **a:x** is the list obtained by adding **a** as first list element to list **x**.

(iv) **x++y** is the result of concatenating lists **x** and **y**.

(v) **x--y** is the difference of lists **x** and **y**: it is unclear to me how this is defined in Miranda, the on-line manual gives no information: the article [T84] is not in accordance with the implementation I used. What actually happens is the following: **[1,2,3,1,2,3]--[3,2] = [1,1,2,3]**.

(vi) **[1..10]** is the list containing the numbers **1** through **10**.

(vii) **[1,6..104]** is the list containing the arithmetic series **1, 6, 11, ..., 96, 101**.

(viii) Infinite lists like **[1..]** and **[1,5..]** are allowed.

(ix) Set-notation is accepted: e.g. **[n*m | n<-[11,22..]; m<-[1..100]]**, where **<-** is a typographical attempt at $\in$.

(x) Also: **[a |(a,b)<-(1,1),(b,a+b)..]**, which denotes the list of Fibonacci numbers.

(xi) Sets are treated as lists without duplications.
example **{[a,b] | a,b<-[1..]; a-b=5}**

2.3 In Miranda there is no notion of program

Working in Miranda is different from programming in an old-fashioned von Neumann language like Algol-68. The user of Miranda creates herself an environment of declarations of types and definitions of functions, called *the script*. In a typical session (on-line in a UNIX-setting) Miranda is called upon and the desired script is compiled. Then the user evaluates whatever expression she likes.
Example: take as script the Miranda fragment proposed in 2.1.1. When the script is compilated one can proceed with

add_1 [1,2,3,4] [4,3,2,1] i.e., $(1+2x+3x^2+4x^3)+(4+3x+2x^2+x^3)$

and Miranda will evaluate this expression and answer

[5,5,5,5] i.e., $(5+5x+5x^2+5x^3)$

3. The Fast Fourier Transform

Unthoughtful multiplication of two polynomials of degree n will in most cases lead to a computation costing some $O(n^2)$ operations. Application of elementary complex number theory and the method of the Fast Fourier Transform results in a divide-and-conquer algorithm costing only $O(n \log n)$ operations. We will give a short explanation (adopted from Sedgewick's [S83]) of this method.

Let us take in mind two polynomials **f** and **g** with complex coefficients. Let **N-1** be the degree of the product **h** of **f** and **g**.

It is an well-known fact that one can uniquely determine all **N** coefficients of **h**, if one knows the value of **h** on **N** different inputs. We get such a collection of output values of **h** if we evaluate **f** and **g** and multiply the results for each of the **N** inputs.

The crucial idea of the algorithm is that one uses the **N** complex **N**-th roots of unity as set of input values. On such privileged sets of inputs a fast divide-and-conquer algorithm is available for the evaluation of polynomials. If one employs the method of the Fast Fourier Transform, then essentially the same algorithm can be used for the inverse problem, interpolation.

3.1. Evaluation of a polynomial of degree 2^k-1

Let $f(x)=a_0+a_1x+...+a_{N-1}x^{N-1}$ be a polynomial of degree **N-1** for $N=2^k$. We want to evaluate **f** on all N-th roots of unity: $w_{0,N}=1, w_{1,N}, ..., w_{N-1,N}$ in the usual cyclic order.

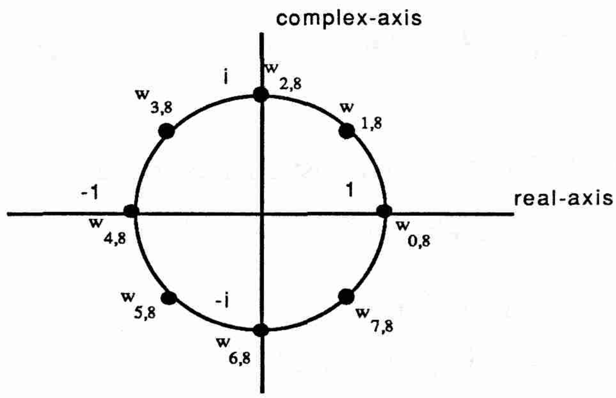


Figure containing all 8-th roots of unity

We want to divide **f** into two polynomials, one containing the even coefficients and the other containing the odd coefficients: we define

$$f_{even}(x)=a_0+a_2x+...+a_{N-2}x^{(N-1)/2}$$
$$f_{odd}(x)=a_1+a_3x+...+a_{N-1}x^{(N-1)/2},$$

then

$$f(x)=f_{even}(x^2)+xf_{odd}(x^2).$$

Hence to evaluate **f** on all **N**-th roots it suffices to evaluate both f_{even} and f_{odd}

on only $w_{0,N}=1$, $w_{2,N}$, $w_{4,N}$, ..., $w_{1/2N-1,N}$, i.e., on all 2^{k-1}-th roots of unity.

The recurrence stops, of course, when we reach the roots **1** and **-1**, since then evaluation is simply a matter of addition of complex numbers. If **C(N)** is the cost (say additions and multiplications) it takes to evaluate a polynomial of degree **N-1** on all **N**-th roots, then clearly $C(N)=2C(N)+2N$, that is $C(N)=O(N\log N)$.

3.2. Interpolation of a polynomial of degree 2^k-1

Again, let $f(x)=a_0+a_1x+...+a_{N-1}x^{N-1}$ be a polynomial of degree **N-1** for $N=2^k$.

Let $N=2^k-1$, and let for each **N**-th root $w_{i,N}$ a complex number b_i be given. The aim of *interpolation* is to find a polynomial $f(x)=a_0+a_1x+...+a_{N-1}x^{N-1}$ of degree **N-1** such that $f(w_{i,N})=b_i$ for all $0\leq i\leq N-1$.

To understand the following calculation note that:

$$(w_{0,N}) = 1$$

$$(w_{m,N})^N = 1$$

$$(w_{m,N})^{-1} = w_{-m,N}$$

$$(w_{m,N})^j = (w_{1,N})^{mj} = (w_{j,N})^m$$

$$\Sigma_j\ (w_{i-m,N})^j = ((w_{i-m,N})^N-1)\ /\ (w_{i-m,N}-1) = 0,\ \text{if}\ i-m\neq 0.$$

The method of the Fast Fourier Transform proceeds as follows.

Let $g(x)=b_0+b_1x+...+b_{N-1}x^{N-1}$,

then
$$\begin{aligned} g(w_{-m,N}) &= \Sigma_j\ b_j(w_{m,N})^{-j} \\ &= \Sigma_j\ f(w_{j,N})(w_{m,N})^{-j} \\ &= \Sigma_j\ \Sigma_i\ a_i(w_{j,N})^i\ (w_{j,N})^{-m} \\ &= \Sigma_j\ \Sigma_i\ a_i(w_{j,N})^{i-m} \\ &= \Sigma_i\ a_i\ \Sigma_j\ (w_{j,N})^{i-m} \\ &= \Sigma_i\ a_i\ \Sigma_j\ (w_{i-m,N})^j \quad \text{(provided } i>m\text{, the other case is similar)} \\ &= Na_m \end{aligned}$$

That is, the coefficients a_m of the sought polynomial **f** can be found by evaluating **g** at the inverse of $w_{m,N}$ and dividing the result by **N**.

Since the cost of evaluating a polynomial on all **N**th-roots is $O(N \log N)$, it follows that interpolation is of the same order.

3.3 The algorithm for multiplication of two polynomials

The algorithm for the multiplication of two complex polynomials **f** and **g** proceeds now as follows:

(i) calculate the smallest **N** of the form $2^k \geq 1$+degree **f.g**,

(ii) evaluate **f** and **g** at the **N**-th roots of unity,

(iii) multiply the values found for each root,

(iv) interpolation on the resulting values.

The cost of this algorithm is clearly

$$O(\text{N log N}) = O((\text{degree f + degree g})\log(\text{degree f + degree g}))$$

(where **N** is the smallest $2^k \geq 1$+degree **f.g**)

3.4 The Miranda script for multiplication of two polynomials

We give the script that contains the definition of the function **multiply**, that multiplies two complex polynomials. **Multiply** is built up from other functions, that will be explained in the next paragraphs:

```
multiply f g = interpolate (mult (bieval (extend (degree (f,g)))))
```

3.4.1 The complex numbers and some useful functions in Miranda

```
complex == (num,num)
```
the type complex consists of pairs of real numbers

```
i = (0,1)
ca,cs,cm :: complex->complex->complex
ca (a,b) (c,d) = (a+c,b+d)
```
complex addition

```
cs (a,b) (c,d) = (a-c,b-d)
```
complex subtraction

```
cm (a,b) (c,d) = (a*c-b*d,b*c+a*d)
```
complex multiplication

```
root :: num->num->complex
```
calculates the j-th n-th root of unity

```
root j n = f(g(j,n))
           where
             f z = (cos z,sin z)
             g(j,n) = 2*pi*j/n
```

```
allroots :: num->poly
```
allroots will contain all n-th roots of unity

```
allroots n = [root j n | j<-[0..n-1]]
```

```
select :: poly->num->poly
```
Intended use: select($[a_1,\ldots,a_{2n}]$,2n):= $[a_1,a_3,\ldots,a_{2n-1}]$

```
select (roots,n) = [roots!(2*j-1)| j<-[1..n/2]]
```

3.4.2 Polynomials and some useful functions in Miranda

```
poly == [complex]
```
poly(nomials) are sequences of complex numbers

```
pa :: poly->poly->poly
pa f [] = f
pa [] g = g
pa (a:f) (b:g) = (a  $ca b):(f $pa g)
```
polynomial addition

```
even :: poly->num->poly
even f n = [f!n | n<-[1,3..n]]
```
takes all even coefficients of f when n is length of polynomial f

```
odd :: poly->num->poly
odd f n = [f!n | n<-[2,4..n]]
```
takes all odd coefficients of f when n is length of polynomial f

```
multarray :: poly->poly->poly
```
multarray $([a_1,...,a_n][b_1,...,b_n])$:= $[a_1b_1,...,a_nb_n]$

```
multarray ([],[]) = []
multarray (a:f) (b:g) = (a $cm b):(multarray (f,g))
```

Note that the behavior of **multarray** is undefined on arrays of different length.

3.4.3 Step 1 of the algorithm 3.3

From now on we will skip the type declarations

```
power :: num->num
power n = 1, n<=1
        = 2*power(n/2)
```
power (n) = least $2^k \geq n$

input will be of the form 2^k

```
degree (f,g) = (f,g,power(#f+#g-1))
```
If f and g are polynomials of respectively degree n-1 and m-1, then #f=n and #g=m and the degree of f.g equals n+m-2, which equals 1 + degree f + degree g.

```
extend (f,g,n) = (f $pa o n, g $pa o n, n, allroots n)
                 where
                 o n = [(0,0)| i<-[1..n]]
```
This function performs three parallel tasks: if n>length f and n>length g, then it extends both f and g with 0, such that the length of the new polynomials is n, and it makes a list containing all n-th roots.

Observe that if Miranda evaluates the expression **extend(degree(f,g))**, then it performs step 1 of the algorithm descibed above in 3.3.

3.4.4 Step 2: evaluation of f and g

We will need the following function **mix** in our description of **evaluate**.

```
mix(fev,fod,n,roots) = [(fev!i) $ca ((roots!i) $cm (fodd!i)) |i<-[1..n/2]]++
                       [(fev!i) $ca ((roots!(n/2+i)) $cm (fodd!i)) |i<-[1..n/2]]
```

The intended use of mix is:

$mix(f_{even}, f_{odd}, N, [w_{0,N}, \ldots, w_{N-1,N}]) =$

$$[f_{even}(w_{0,N}^2)+w_{0,N}f_{odd}(w_{0,N}^2),\ldots,f_{even}(w_{N-1,N}^2)+w_{N-1,N}f_{odd}(w_{N-1,N}^2)]$$

That is, it makes a list of all values of $\mathbf{f(x)=f_{even}(x^2)+xf_{odd}(x^2)}$ (cf. 3.1) when x runs through all N-th roots. Note that x^2 runs twice as fast as x, this explains the second row of the definition of mix.

```
evaluate (f,n,roots) = [(0,0)], n<=0
                     = f, n=1
                     = [(f!1)$ca(f!2), (f!1)$cs(f!2)], n=2
                     = mix (feven,fodd,n,roots)
                       where
                       feven = evaluate (even f n, n/2, select(roots,n))
                       fodd = evaluate (odd f n, n/2, select(roots,n))
```

The intended use of evaluate is:

$$evaluate\ (f,N,[w_{0,N},\ldots,w_{N-1,N}]) = [f(w_{0,N}),\ldots,f(w_{N-1,N})].$$

Evaluation of the function **evaluate** results in the recurrence described in 3.3.

```
bieval (f,g,n,roots) = (evaluate (f,n,roots), evaluate (g,n,roots))
```

Evaluation of the function **bieval** performs step 2 of the algorithm in 3.3.

3.4.5 Step 3: multiplication of the values found for each root

```
mult (f,g,n,roots) = (multarray(f,g), n, roots)
```

No explanation necessary... .

3.4.6 Step 4: interpolation on the resulting values

```
divide [] n = n                                all elements of the list of complex
                                               numbers are divided by n
divide ((a,b):f) n = (a/n,b/n):(divide f n)
```

div_and_rev ([],n) = [] div_and_rev($[a_0,a_1,\ldots,a_m]$,n):= $[a_0/n,a_m/n,\ldots a_1/n]$

div_and_rev ((a,b):f,n) = (a/n,b/n):(reverse divide f n))

Intended use is on the list of allroots, recall that $[w_{0,N},w_{-1,N},\ldots,w_{-(N-1),N}] = [w_{0,N}, w_{N-1,N}, \ldots,w_{1,N}]$, which explains for the strange reversing of the order.

interpolate (f,n,roots)= div_and_rev (evaluate(f,n,roots), n)

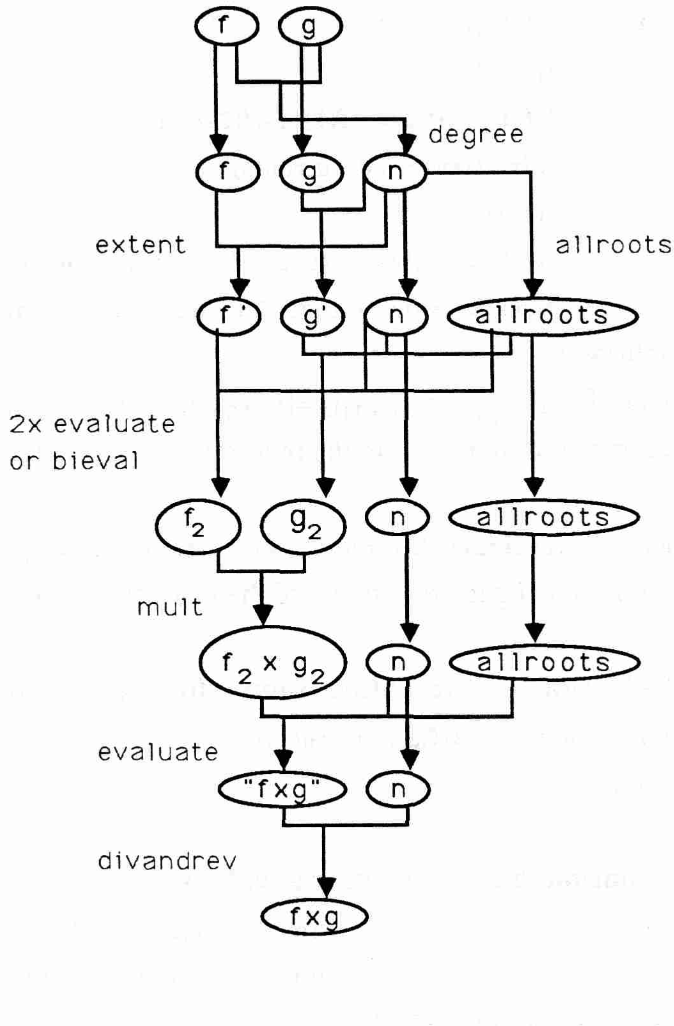
Intended use:

interpolate $(f,N,[w_{0,N},\ldots,w_{N-1,N}]) := [f(w_{0,N})/N, f(w_{-1,N})/N,\ldots,f(w_{-(N-1),N})/N]$

3.4.7 Finally the multiplication function

multiply f g = interpolate (mult (bieval (extend (degree (f,g)))))

Combine all previous steps: the definition of multiply follows the strategy of the algorithm as outlined in 3.3.



Data Flow Diagram of Multiplication Algorithm

4. Observations on Miranda programming

4.1 How to pass on already calculated results

The reader of the above program wondered maybe why the function **multiply** has only two polynomials as input, while all the defining sub-functions map long tuples of attributes in other long tuples. Look at **evaluate**, for instance. The reason becomes clear when one considers the Data Flow Diagram of the previous page. For example, from the input polynomials **f** and **g** the algorithm calculates a number **n** and from this n a list of all n-th roots of unity, **allroots**. Both are needed as input in several following calculations of the algorithm. So the function **evaluate** gets **n** and **allroots** as input, uses them in the actual calculation, and passes them on together with the answer of the calculation, so that the following functions do not need to recalculate **n** and **allroots** from scratch.

Note, however, that this parameter mechanism is not basically different from the parameter mechanism in a von Neumann programming language like Pascal.

4.2 Is functional programming good for writing structured software?

We claim that functional programming -for instance in Miranda- is indeed good for writing structured software. The above script may serve as an example. We can give several reasons:

(i) the notational ease with which one can introduce types and the functions one needs on those types. Writing such a script resembles - so it feels - very much the writing of a mathematical text. One introduces the concepts in the natural order, i.e., following the flow of the argument. Here: first complex numbers and related functions are introduced, then the complex polynomials and their allies.

(ii) the ease with which one can handle lists in combination with the simple way to define functions recursively enables one to define things sometimes surprisingly short and conspicuous. Turner gives several examples of this [T84]. (My favorite is the following definition of the list of all primes:

```
primes  = sieve [2..]
          where
          sieve (p:x) = p:sieve[n<-x | n rem p ~=0])
```

Our program contains other examples. Compare for instance the definition of the function **evaluate** with the Pascal procedure **eval** in [S83] (cf. appendix). Maybe this comparison is a bit unfair: the Pascal procedure is cleverly written, such that no more memory is used then strictly necessary, the recurrence steps do not consume their own additional memory: everything is performed in one array. This feature is lost in our

Miranda program, we don't know how cheap or expensive the script is implemented with respect to memory. The naive idea is that one does not need to worry so much about memory. Moreover, one can trust the implemented garbage collector to take care of the wasted bits of memory.

(iii) A programmer should strive for a readable script, i.e. a not too complicated script. My personal experience with programming in Miranda is that the language does not seem to allow for complicated scripts. The reason seems to be that in Miranda one can only define types and functions with only a few and elementary expressions, so that one is almost forced to define simple functions with a clearly described content, that easily can be tested on intended inputs.

Whether functional programming is better then programming in von Neumann style programming languages, I can not say. There is a large deal of personal taste involved in such a matter. And the subject of the Fast Fourier Transform can turn out been a too simple test. The important thing is, however, that one can write well-structured and easily readable software in a functional programming language like Miranda.

4.3 Does functional programming allow for parallel evaluation?

It seems to me that the right statement is that writing a functional program for a particular algorithm forces one to get a clear idea of the flow of the data through the algorithm. This can result in an explicit data flow chart of the algorithm, like the one on the previous page. And one can easily see in the diagram which parts of the algorithm can be processed in parallel.

However, just from writing Miranda scripts one cannot learn that functional programming makes parallel evaluation possible. That is something which has to do with the implementation of Miranda. We feel that it has to be an implementator of Miranda to judge the possibility of parallel evaluation of Miranda scripts, and to compare the claimed relative ease of such a implementation with the difficulties one does, or does not encounter when one tries to implement a von Neumann-style language such that parallel evaluation is possible.

5. Conclusion

Returning to the two claims on quality investigated by functional programmers mentioned in the introduction, we can say that:

(i) A functional programming language like Miranda is a good medium to develop structured software in. The resulting scripts, or programs, seem to be shorter and more conspicuous compared with similar programs in Pascal. On the other hand we have tested only one algorithm.

(ii) Writing in a functional programming language like Miranda forces one to get a clear mental picture of the flow of data through the algorithm, but the same should also happen to, let us say, a good Pascal programmer. Analysis of this flow chart will reveal which part of the algorithm can be processed in parallel.

6. Literature

[BvL] Barendregt, Henk, & Marc van Leeuwen, *Functional programming and the language TALE,* preprint 412, Dept. of Math., Univerity of Utrecht, 1986, pp.87.

[T84] Turner, D.A., *Functional programs as executable specifications,* Phil. Trans. R. Soc. Lond. A 312, 363-388 (1984).

[T85] Turner, D.A., *Miranda: A non-strict functional language with polymorphic types,* in: proceedings IFIP international conference on functional programming languages and computer architecture, Nancy , September 1985, (Springer Lecture Notes in Computer Science, 201)

[S83] Sedgewick, Robert, *Algorithms,* Addison-Wesley Publishing Company, Reading Massachusetts etc, 1983.

Appendix 1

(procedure taken from Sedgewick's *Algorithms*.)

```
procedure eval(var p: poly; N, k: integer);
  var i, j: integer;
  begin
  if N=1 then
    begin
    t:=p[k]; p1:=p[k+1];
    p[k]:=t+p1; p[k+1]:=t-p1
    end
  else
    begin
    for i:=0 to N div 2 do
      begin
      j:=k+2*i;
      t[i]:=p[j]; t[i+1+N div 2]:=p[j+1]
      end;
    for i:=0 to N do p[k+i]:=t[i];
    eval(p, N div 2, k);
    eval(p, N div 2, k+1+N div 2);
    j:=(outN+1) div (N+1);
    for i:=0 to N div 2 do
      begin
      t:=w[i*j]*p[k+(N div 2)+1+i];
      t[i]:=p[k+i]+t; t[i+(N div 2)+1]:=p[k+i]-t
      end;
    for i:=0 to N do p[k+i]:=t[i]
    end
  end;
```

Appendix 2

A Functional Program for the Fast Fourier Transform.

The *italic* fragment describes roughly the same as the PASCAL procedure of appendix 1.

```
complex == (num,num)

i = (0,1)
ca,cs,cm :: complex->complex->complex
ca (a,b) (c,d) = (a+c,b+d)
cs (a,b) (c,d) = (a-c,b-d)
cm (a,b) (c,d) = (a*c-b*d,b*c+a*d)

root :: num->num->complex
root j n = f(g(j,n))
          where
      f z = (cos z,sin z)
        g(j,n) = 2*pi*j/n

allroots :: num->poly
allroots n = [root j n | j<-[0..n-1]

poly == [complex]

pa :: poly->poly->poly
pa f [] = f
pa [] g = g
pa (a:f) (b:g) = (a  $ca b):(f $pa g)

multarray :: poly->poly->poly
multarray ([],[]) = []
multarray (a:f) (b:g) = (a $cm b):(multarray (f,g))

power :: num->num
power n = 1, n<=1
        = 2*power(n/2)

degree (f,g) = (f,g,power(#f+#g-1))

extend (f,g,n) =  (f $pa o n, g $pa o n, n, allroots n)
                  where
                  o n = [(0,0)| i<-[1..n]]


even :: poly->num->poly
even f n = [f!n | n<-[1,3..n]]

odd :: poly->num->poly
odd f n = [f!n | n<-[2,4..n]]

select :: poly->num->poly
select (roots,n) = [roots!(2*j-1)| j<-[1..n/2]]

mix(fev,fod,n,roots) = [(fev!i) $ca ((roots!i) $cm (fodd!i)) |i<-[1..n/2]]++
```

```
[(fev!i) $ca ((roots!(n/2+i)) $cm (fodd!i)) |i<-[1..n/2]]

evaluate (f,n,roots)  = [(0,0)], n<=0
                      = f, n=1
                      = [(f!1)$ca(f!2), (f!1)$cs(f!2)], n=2
                      = mix (feven,fodd,n,roots)
                        where
                        feven = evaluate (even f n, n/2, select(roots,n))
                        fodd = evaluate (odd f n, n/2, select(roots,n))

bieval (f,g,n,roots) = (evaluate (f,n,roots), evaluate (g,n,roots))

mult (f,g,n,roots) = (multarray(f,g), n, roots)

divide [] n = n
divide ((a,b):f) n = (a/n,b/n):(divide f n)

div_and_rev ([],n) = []
div_and_rev ((a,b):f,n) = (a/n,b/n):(reverse divide f n))

interpolate (f,n,roots)= div_and_rev (evaluate(f,n,roots), n)

multiply f g = interpolate (mult (bieval (extend (degree (f,g)))))
```