Deriving the fast Fourier algorithm by calculation

Geraint Jones
Programming Research Group
11 Keble Road
Oxford OX1 3QD

Abstract

This paper reports an explanation of an intricate algorithm in the terms of a potentially mechanisable rigorous-development method. It uses notations and techniques of Sheeran [1] and Bird and Meertens [2, 3]. We have claimed that these techniques are applicable to digital signal processing circuits, and have previously applied them to regular array circuits [4, 5, 6].

This paper shows that they can deal with an apparently very different and more complex algorithm: the fast Fourier transform. Similar papers to this one [7, 8, 9] perform most of the same calculations, but experiment with different ways of expressing the algorithms and their development.

Twenty-five years ago Cooley and Tukey rediscovered an optimising technique usually attributed to Gauss, who used it in hand calculation. They applied the technique to the discrete Fourier transform, reducing an apparently $O(n^2)$ problem to the almost instantly ubiquitous $O(n \log n)$ 'fast Fourier transform' [10]. The fast Fourier transform is not of course a different transform, but a fast implementation of the discrete transform.

Its greatest virtue lies in that it can be executed in $O(\log n)$ time on $O(n)$ processors in a uniform way – it lends itself to a low-latency high-throughput pipelined hardware implementation. Indeed, a footnote to the Cooley–Tukey paper records that a hardware implementation was underway as the paper was published, specifically that a component for evaluating a four-point transform had been 'designed by R. E. Miller and S. Winograd of the IBM Watson Research Centre'.

The unfortunate disadvantage of the fast algorithm is that although the fundamental idea is simple, the detail of its efficient implementation is very hard to understand. That efficiency depends on intricate permutations which rearrange data to maximise the sharing of work done in calculating intermediate results. Presentations of the algorithm abound in mysterious artefacts like the reversal of bits in subscripts [11], and the translation of parts of subscripts from time space to frequency space [12]. More recent descriptions of implementations seem to gloss over the problem, either referring the reader back to older presentations [13], or apparently assuming that the algorithm – because it is well known – must be well understood [14].

This paper reports some success in describing the derivation of the the Cooley–Tukey fast Fourier algorithm from the specification of the discrete Fourier transform.

A functional programming notation was used to express the discrete transform, and the fast algorithm has been calculated from it by equational reasoning. The calculation has been carried out in some detail as part of the feasibility study for a mechanical circuit-designer's assistant. The style of the calculation is such that we believe that the process of deriving a reasonable layout of an implementing circuit from our final program would also be mechanisable.

The discrete Fourier transform

The discrete Fourier transform is defined in terms of the arithmetic on an integral domain. You can think of arithmetic on complex numbers, for a definite example, although there are applications where finite fields or vector spaces over integral domains are appropriate. The derivation depends only on the algebraic properties of the arithmetic, not on the underlying arithmetic itself, so everything said here about the algorithm will be true for finite fields and vector spaces as well.

The discrete Fourier transform of a vector x of length n is a vector y of the same length for which

$$y_j \quad = \quad \sum_{k:0\leqslant k<n} \omega^{j\times k} \times x_k$$

where ω is a principal n-th root of unity. (In the example of complex numbers, you can think of $\omega = e^{2\pi i/n}$.) The result, y, is sometimes called the 'frequency spectrum' of the sample x.

Even if the powers of ω are pre-calculated, it would appear that $O(n^2)$ multiplications are required to evaluate the whole of y for any x. The fast algorithm avoids many of these by making use of the fact that $\omega^n = 1$. The discovery made by Cooley and Tukey was that if n is composite, the calculation can be divided into what amounts to a number of smaller Fourier transforms. Suppose $n = p \times q$, then by a change of variables

$$\begin{aligned}
y_{pa+b} \quad &= \quad \sum_{c:0\leqslant c<p} \ \sum_{d:0\leqslant d<q} \omega^{(pa+b)(qc+d)} x_{qc+d} \\
&= \quad \sum_{c:0\leqslant c<p} \ \sum_{d:0\leqslant d<q} (\omega^{pq})^{ac} (\omega^{p})^{ad} (\omega^{q})^{bc} \omega^{bd} x_{qc+d} \\
&= \quad \sum_{d:0\leqslant d<q} (\omega^{p})^{ad} \omega^{bd} \sum_{c:0\leqslant c<p} (\omega^{q})^{bc} x_{qc+d}
\end{aligned}$$

Since ω^q is a p-th root of unity, and ω^p is a q-th root of unity, it is not surprising that the above calculation leads to an implementation in which p-sized and q-sized transforms appear. It is harder, however, to see what that implementation might be.

A notation for describing circuits

To simplify calculation with algorithms, we write and work with expressions which represent not data values, but functions. This requires a variety of operators for combining functions – algorithms, or circuits, depending on whether you think we are designing programs or hardware – rather than the more usual operators which act on data.

The basic operation on circuits is composition, $f \cdot g$ defined by $(f \cdot g)x = f(gx)$, which you can think of as connecting the output of g to the input of f. We have previously [7] tried explaining this development in terms of 'reverse composition', $f ; g = g \cdot f$, which is easier to read left-to-right as an operational description, but which fits less well with our other notational conventions.

All our data are organised in finite lists – or vectors – and many of the operators in this paper describe the way that a function operating on a signal is manipulated to make a function which operates on a list of signals. In this way we avoid having to manipulate subscript expressions or individual components of the vectors. For example, $f * x$ read 'f map x', is defined by $(f * x)_i = fx_i$ and represents the replication of the circuit f so that each instance can be connected to one of each of the signal sources in the list x. We write $f*$, or sometimes $(f*)$ for the replicated circuits – the function that takes the list x as input and returns the list $f * x$. Similarly, $f**$ means $(f*)*$ which is a circuit that expects a list of lists of signals – a column of rows, say – and applies $f*$ to each row, which is to say that it applies f to each element of each row. Very occasionally we will be driven to write $(*)$ for the function which when applied to f returns $f*$, that is $(*)f = f*$.

One of the more useful properties of map is that it distributes over composition: $(f \cdot g)* = f* \cdot g*$ irrespective of the particular functions f and g. Moreover, if f and g commute, that is if $f \cdot g = g \cdot f$, then $f* \cdot g* = (f \cdot g)* = (g \cdot f)* = g* \cdot f*$ so $f*$ and $g*$ commute, and so do $f**$ and $g**$, and so on. All our calculations are of essentially this form, and rely on a rich collection of laws none of them significantly more complex than these.

The concatenation of lists, $x \mathbin{+\!\!+} y$ is the list consisting of the elements of x followed by those of y. (All our lists are of finite length.) A function like $f*$ is called a homomorphism of lists because $f * (x \mathbin{+\!\!+} y) = (f * x) \mathbin{+\!\!+} (f * y)$. Homomorphisms are clearly ideal candidates for parallel implementation.

Reduction is a generalisation of the way that the $\sum$ operator applies addition to a list of values. We write $\oplus / x$, read 'the $\oplus$ reduce of x', for the value of $x_0 \oplus x_1 \oplus x_2 \oplus \ldots$ but only when $\oplus$ is associative so that it does not matter in what order the $\oplus$ operations are applied. (We follow a convention of Bird and Meertens that the symbols $\oplus$, $\otimes$ and so on are not specific operators but are usually operator variables, just as f, g and so on are function variables, and x, y and so on are data variables.) Flatten, $\mathbin{+\!\!+}/$, is the operation which joins a list of lists to make a single list, and the generalisation of the homomorphism property of map is that $f* \cdot \mathbin{+\!\!+}/ = \mathbin{+\!\!+}/ \cdot f**$. In the case of the usual arithmetic operations $+$ and $\times$, for which $n \times (x + y) = (n \times x) + (n \times y)$, it is similarly the case that $n\times \cdot +/ = +/ \cdot n \times *$. This is called the distribution of $\times$ over $+$, so we will also say that $*$ distributes over $\mathbin{+\!\!+}$.

For any operation $\oplus$, its reduction can be divided over $\mathbin{+\!\!+}$ because $\oplus / (x \mathbin{+\!\!+} y) = (\oplus / x) \oplus (\oplus / y)$, and more generally $\oplus/ \cdot \mathbin{+\!\!+}/ = \oplus/ \cdot \oplus / *$. That means that you can reduce a list of lists either by concatenating the rows and reducing the whole, or by reducing each of the rows and then reducing the list of results. The equality $\oplus/ \cdot \mathbin{+\!\!+}/ = \oplus/ \cdot \oplus / *$ therefore captures the essence of the associativity of the $\oplus$ operation.

If x and y are lists of the same length $x \Upsilon_{\oplus} y$, read as 'x zip-with-$\oplus$ y', is the point-by-point operation defined by $(x \Upsilon_{\oplus} y)_i = x_i \oplus y_i$. This is another replication operation like map, which produces an operation that has a naturally parallel implementation. An example is Υ_{+} which is the usual point-by-point addition of vectors.

Some notation specific to this algorithm

In the course of the calculation of the fast Fourier algorithm, we identified a number of useful operations which may not be familiar to the users of the Bird–Meertens calculus. Much of the work of the development is encapsulated in the algebra of these operations.

The transposition of lists of lists is here written И, defined by $(\text{И} x)_{ij} = x_{ji}$. (You can calculate many of its properties from Bird's and Meertens' observation that $\text{И} = \Upsilon_{+\!\!+}/ \cdot 1 \copyright * *$ where $1 \copyright x$ is the list of length one whose only element is x.) Throughout the theory underlying this paper, lists of lists have to be 'rectangular' with every sublist having the same length. In that case it follows from the definitions that for any binary $\oplus$, $\Upsilon_{\oplus}/ = \oplus / * \cdot \text{И}$. That is, if given a column of rows, you can $\Upsilon_{\oplus}$ the rows together column by column by transposing the rows and columns, and then $\oplus$-ing along each of the rows. Consider now

$$
\begin{aligned}
\Upsilon_{+}/ \cdot n\times * * \quad &= \{\text{ transposition rule for } \Upsilon_{+}/ \ \}\\
&\quad +/* \cdot \text{И} \cdot n\times * *\\
&= \{\text{ map acts pointwise, so } \text{И} \cdot f * * = f * * \cdot \text{И} \ \}\\
&\quad +/* \cdot n\times * * \cdot \text{И}\\
&= \{\text{ map distributes over composition }\}\\
&\quad (+/ \cdot n \times *)* \cdot \text{И}\\
&= \{\text{ multiplication distributes over addition }\}\\
&\quad (n\times \cdot +/)* \cdot \text{И}\\
&= \{\text{ map distributes over composition }\}\\
&\quad n \times * \cdot +/* \cdot \text{И}\\
&= \{\text{ transposition rule for } \Upsilon_{+}/ \ \}\\
&\quad n \times * \cdot \Upsilon_{+}/
\end{aligned}
$$

so showing that $n \times *$, which is the multiplication of vectors by the scalar n, distributes over Υ_{+}, which is the point-by-point addition of vectors. The calculation shows that this is a consequence of the distribution property of scalar addition and multiplication.

Transposition is useful in capturing other properties of operations. For example, if as well as being associative $\oplus$ is a commutative operation – that is if $x \oplus y = y \oplus x$ – you can choose to $\oplus$-reduce an array of values either by rows and then columns, or by columns and then rows, that is $\oplus/ \cdot \Upsilon_{\oplus}/ = \oplus/ \cdot \oplus / * \cdot \text{И} = \oplus/ \cdot \oplus / *$. This equality captures the essence of the commutativity of $\oplus$. (Bird describes operators for which $\oplus/ \cdot \Upsilon_{\otimes}/ = \otimes/ \cdot \oplus / *$ by saying that $\oplus$ *abides* with $\otimes$, so commutative operators are ones which abide with themselves.)

Some of the properties of the $*$ operation are shared by operators that have previously usually been used to explain the skewing of data in time [15, 16]. The triangle operation is defined by $(f \bigtriangleup x)_i = f^i x_i$ where f^i represents i repeated applications of f, and the block operation $(f \,\square\, x)_i = f^{\# x} x_i$ where $\# x$ is the length of x. We will call $*$, $\bigtriangleup$, and $\square$ *pointwise* operations.

If $\oplus$ and $\otimes$ are any two pointwise operators, and if f and g are two functions that commute – that is, if $f \cdot g = g \cdot f$ – then so do $f\oplus$ and $g\otimes$ – that is, $f\oplus \cdot g\otimes = g\otimes \cdot f\oplus$ – and by repeated application of this same theorem, so do any pair of terms like $f \bigtriangleup * \bigtriangleup *$ and $f \,\square \bigtriangleup \bigtriangleup *$. Again, if f and g commute and $\oplus$ is pointwise, then $(f \cdot g)\oplus = f\oplus \cdot g\oplus$ and so on, which is reminiscent of the distribution of $*$ over composition. If $\oplus$ and $\otimes$ are pointwise operations, $f \oplus \otimes \cdot \text{И} = \text{И} \cdot f \otimes \oplus$, which is a general form of an earlier observation that $f ** \cdot \text{И} = \text{И} \cdot f **$.

Of course, not all the properties of $*$ are shared by pointwise operations, and it is possible to relax some of the preconditions of these results if it is known that one of the pointwise operators is map. Given functions which do not quite commute, say $f \cdot g = g \cdot h$, then for any pointwise $\oplus$ operation $f\oplus \cdot g* = g* \cdot h\oplus$ even if $f \neq h$.

To construct constant lists $n \copyright x$, read 'n copies of x', is a list of length n, each element of which is x. Although it is an apparently unusual operation, it has properties which are familiar-looking when cast in algebraic terms, for $(m+n) \copyright x = (m \copyright x) +\!\!+ (n \copyright x)$ and $n \copyright (x \oplus y) = (n \copyright x) \Upsilon_{\oplus} (n \copyright y)$. For any f it is clearly the case that $n\copyright \cdot f = f* \cdot n\copyright$ because if you want n copies of the output of a circuit, you can just as easily fan out the output of one instance of the circuit or fan out the input to a number of copies of the circuit. Although you might expect only to use this equation to optimise by replacing the right-hand side by the left, it can also be used left to right so as to increase the amount of parallelism, perhaps in the hope that the $f*$ term can be combined with some later processing to achieve a global simplification.

Because transposition is effectively an interleaving operation, it interacts quite regularly with the copying operator. Copying a list and interleaving the copies is the same as making individual copies of the elements of the list, $\text{И} \cdot n\copyright = n \copyright *$, and replicating the rows of a transposed list of lists is related to replicating the entire list by $n \copyright * \cdot \text{И} = \text{И} \cdot n\copyright \cdot \text{И} = \text{И} \cdot \text{И}* \cdot n\copyright$. Other interactions between these operations can be calculated from these and from earlier equations.

Calculating with functions of specific types

The width (or period) of a Fourier transform is of course a part of the calculation. It transpires that in reasoning about the algorithm, say $\mathcal{F}$, it is necessary to be able to refer this width, which is of course the length of the argument, x. However, the argument does not normally appear in the calculations, which deal with $\mathcal{F}$ rather than with $\mathcal{F}x$ which is the value of the output. There are two apparent alternative techniques for dealing with this. One possibility which suggests itself is to handle the width information as a part of the type of the expression, and perform a parallel calculation of the type alongside the manipulation of the algorithm-valued expression.

The other possibility – explored in reference [7] – is to code the type, where necessary, by introducing functions which are the identities on just that type. This technique makes the type-calculation uniform with the algorithm-calculation, and is probably the right approach to use in implementing mechanical tools to support the calculation. On the other hand, it makes the formulae appear rather strange, and leads to some unnatural manipulations.

Most exponents of this sort of calculation tend to be rather vague about details of type, for example writing the name of a polymorphic function in a calculation even when the calculation is valid only for particular instances. Their calculations are usually supported by an informal natural-language commentary about the restrictions. In this paper, the presentation is a compromise between these approaches: we write type-restrictions in the function-expressions, but will not be quite as careful with the type restrictions as with the values. To do this is simply to cast a cloak of notational formality over the informality of a running commentary.

If f is a function which takes arguments of type α and if β is a subtype of α, the function $f \upharpoonright \beta$ is that which agrees with f but is applicable only to values in β, and to all values in β. Similarly, if f returns values of α' and if β' is a subtype of α', the function $\beta' \upharpoonleft f$ is the largest which agrees with f but returns values in β'. (Small letters from early in the Greek alphabet are type variables in this paper.)

Occasionally $f \cdot g$ is written even when the domain of f is strictly smaller than the range of values returned by g, intending by that to indicate a restriction of g. The cost of this otherwise harmless convention is that even if g is a bijection, it is not necessarily the case that $f = (f \cdot g) \cdot g^{-1}$. On the other hand, it is always the case that $(f \upharpoonright \alpha) \cdot g = (f \upharpoonright \alpha) \cdot (\alpha \upharpoonleft g) = f \cdot (\alpha \upharpoonleft g)$. Moreover, restriction associates with composition, $(f \cdot g) \upharpoonright \alpha = f \cdot (g \upharpoonright \alpha)$ and $\alpha \upharpoonleft (f \cdot g) = (\alpha \upharpoonleft f) \cdot g$, so all the brackets can be left out, and if you prefer you can read the restrictions as compositions with identity functions.

The types that need to be named in this calculation are all the types of zeroth order objects: values of the integral domain over which the arithmetic is defined, lists of these, and lists of lists and so on. The type of lists (of any length), each component of which is of type α, would usually be written α^*, making a pun between the list-type constructor and the operation of mapping over lists. The subtype of that type containing just lists of length n each component of which is of type α, would similarly be written α^n. So, for example, $\alpha^{*,p}$ is the type of p-lists of lists of α values; $\alpha^{q,*}$ is the type of lists of q-lists of α values; and $\alpha^{q,p}$ is the type of p-lists of q-lists of α values. This last type is the greatest common subtype of the preceding two, and in general the greatest common subtype – intersection – of two types satisfies $\alpha^{x,y} \cap \beta^{p,q} = (\alpha \cap \beta)^{(x \sqcup p),(y \sqcup q)}$ where $*$ is now also punned with the bottom of the flat lattice of natural numbers. This gives a way of factoring type restrictions, for example $f \upharpoonright (\alpha \cap \beta) = f \upharpoonright \alpha \upharpoonright \beta$.

The calculations in this paper use a number of rules about the interaction between specific operations and type restriction, for example that $\text{И} \upharpoonright \alpha^{x,y} = \alpha^{y,x} \upharpoonleft \text{И}$, that $n© \upharpoonright \alpha = \alpha^n \upharpoonleft n©$, and that $f\Box \upharpoonright \alpha^n = f^n{*} \upharpoonright \alpha^n$, for any proper natural number n. Moreover, if f is homogeneous on α, that is if $f \upharpoonright \alpha = \alpha \upharpoonleft f$, and if $\oplus$ is pointwise, then $f\oplus$ is homogeneous on α^* and on each α^n.

For the most part, in this paper the only part of a type which is relevant is whether it is a list type, or the number of components in that list. Accordingly we shall usually omit the base type, writing $f \upharpoonright *, p$ for the restriction of f to p-lists of lists; $f \upharpoonright q, *$ for the restriction of f to lists of q-lists; and so on.

Since $\#(x \mathbin{+\!\!+} y) = (\# x) + (\# y)$, it follows that $\# \cdot \mathbin{+\!\!+}/ = +/ \cdot \# *$ and so that $\mathbin{+\!\!+}/ \upharpoonleft n, m = n \times m \upharpoonright \mathbin{+\!\!+}/ \upharpoonleft n, m$. Notice that you cannot in general compare these last two functions with $n \times m \upharpoonright \mathbin{+\!\!+}/$ since they are applicable only to lists of length m with sublists of length n, whereas $n \times m \upharpoonright \mathbin{+\!\!+}/$ can flatten any rectangular list of lists with a total length of $n \times m$, and there will be other factorisations of this product unless n and m are equal primes.

Casting the algorithm in the notation

The first task in a calculation dealing with an algorithm is to cast the specification in the notation that will be used to handle the development. There are two things which we do in this stage.

One part appears to be largely a process of eliminating subscripts, since the usual convention is to specify separately each co-ordinate of an output vector. The conventional understanding of a specification of the form $y_i = \ldots$ is that the subscript is universally quantified, so that this one equation formally represents a number of different equations, one for each value of i. To make clear that an algorithm operates uniformly at all co-ordinates of its output we write a single equation which defines the whole list of output values. This means that we need (temporarily) a notation for lists, which we write $\langle i : 0 \leqslant i < n : x_i \rangle$ for the list of length n, the ith element of which is x_i.

The other part of the translation is to manipulate the specification – which is usually an expression describing the output of a calculation for a given input – into the form of an application to that input of an expression representing the algorithm. The manipulation of the algorithm can then proceed without reference to the particular input.

The discrete Fourier transform was specified by

$$y_j \quad = \quad \sum_{k:0 \leqslant k < n} \omega^{j \times k} \times x_k$$

by which was meant that the output y should be defined for each j in the range $0 \leqslant j < n$, so meaning that

$$\begin{aligned}
y \quad = \quad & \langle j : 0 \leqslant j < n : \textstyle\sum \langle k : 0 \leqslant k < n : \omega^{j \times k} \times x_k \rangle \rangle \\
= \{ & \text{ meaning of summation, meaning of arithmetic exponentiation } \} \\
& \langle j : 0 \leqslant j < n : + / \langle k : 0 \leqslant k < n : (\omega^j)^k \times x_k \rangle \rangle \\
= \{ & \text{ meaning of } * \text{ and associativity of } \times \} \\
& +/ * \langle j : 0 \leqslant j < n : \langle k : 0 \leqslant k < n : ((\omega \times)^j)^k x_k \rangle \rangle \\
= \{ & \text{ meaning of } \triangle \} \\
& +/ * \langle j : 0 \leqslant j < n : (\omega \times)^j \triangle x \rangle \\
= \{ & \text{ distribution of } \triangle \text{ over commuting composition, meaning of } \copyright \} \\
& +/ * \langle j : 0 \leqslant j < n : ((\omega \times) \triangle)^j (n \copyright x)_j \rangle \\
= \{ & \text{ meaning of the } \triangle \text{ operator } \} \\
& +/ * (((\omega \times) \triangle) \triangle (n \copyright x))
\end{aligned}$$

$= \{$ meaning of composition $\}$

$((+/*) \cdot (((\omega\times)\triangle)\triangle) \cdot (n©))x$

$= \{$ conventions about parentheses $\}$

$(+/* \cdot \omega\times \triangle \triangle \cdot n©)x$

Since ω depends on n, because $\omega^n = 1$, we will write $\omega\times$ using a new operator ς for which $n \varsigma z = \omega \times z$. This operation has the property, which will be useful later, that $((p \times q)\varsigma)^q = (p\varsigma)$.

The term $+/* \cdot n\varsigma \triangle \triangle \cdot n©$ represents the discrete Fourier transform algorithm, but is only applicable to lists of length n, so we will calculate from the definition

$$\mathcal{F} \upharpoonright n \quad = \quad +/* \cdot n\varsigma \triangle \triangle \cdot n© \upharpoonright n$$

Dividing large problems into smaller ones

Suppose f is an algorithm or circuit for calculating some list-valued function of a list of values. If it is possible to express f in the form $+\!\!+/ \cdot g$, then g is an algorithm for constructing the same result in parts, and may be implementable by a number of independent parts. For example, $(m \times n)© = +\!\!+/ \cdot m© \cdot n© = +\!\!+/ \cdot n © * \cdot m©$ describes a divide-and-conquer strategy for fanning out a signal $m \times n$ times by first making m copies, and then independently fanning each of those out n times.

Similarly, $f \cdot +\!\!+/$ is an algorithm which constructs the same result as f from a partition of the same input into a rectangular list of lists. If it is possible to 'simplify' $f \cdot +\!\!+/$ into a form which has a parallel implementation, that gives a strategy for dividing the calculation of f. A particularly useful result in the present case is that $f\triangle \cdot +\!\!+/ = +\!\!+/ \cdot f \square \triangle \cdot f \triangle *$ which means that $f\triangle$ can be implemented by a number of (smaller) independent instances of $f\triangle$ and a triangular array of $f\square$ components. (Notice that this equality depends the decision to allow lists of lists only where every sublist has the same length.)

In the course of factorising the discrete Fourier transform, this rule is applied twice to an instance of an expression of the form $f \triangle \triangle$.

$f \triangle \triangle \cdot +\!\!+/ \cdot +\!\!+/ **$

$= \{$ factorising $\triangle$ $\}$

$+\!\!+/ \cdot f\triangle \square \triangle \cdot f\triangle \triangle * \cdot +\!\!+/ **$

$= \{$ factorising $\triangle$ and properties of $*$ and pointwise operators $\}$

$+\!\!+/ \cdot f\triangle \square \triangle \cdot +\!\!+/ ** \cdot (f \square \triangle \cdot f \triangle *) \triangle *$

$= \{$ again $\}$

$+\!\!+/ \cdot +\!\!+/ ** \cdot (f \square \triangle \cdot f \triangle *) \square \triangle \cdot (f \square \triangle \cdot f \triangle *) \triangle *$

$= \{$ commuting pointwise terms $\}$

$+\!\!+/ \cdot +\!\!+/ ** \cdot f \square \triangle \square \triangle \cdot f \triangle * \square \triangle \cdot f \square \triangle \triangle * \cdot f \triangle * \triangle *$

This factorisation corresponds to the two changes of variables in the earlier calculation with summations. Notice that all four of the terms in f commute, because all of the operators in them are pointwise.

Since the f in question is $n\varsigma$, this is the point to observe that some powers of f are going to be cancellable, specifically that

$$\begin{aligned} n\varsigma \,\square\, \triangle \,\square\, \triangle \restriction q,*,p,* &= n\varsigma^n * \triangle * \triangle \restriction q,*,p,* \\ &= 1\varsigma * \triangle * \triangle \restriction q,*,p,* \\ &= 1\varsigma * * * * \restriction q,*,p,* \end{aligned}$$

where 1ς is the identity on the type underlying the arithmetic.

Dividing the discrete Fourier transform

Suppose that $n = p \times q$. The factorisation of the n-point transform proceeds, as suggested above, by simplifying a specific instance of $\mathcal{F} \restriction n \cdot +\!\!\!+/$. The particular instance is chosen – with hindsight, of course – so that a term can be cancelled later.

$$\begin{aligned}
& \mathcal{F} \restriction n \cdot +\!\!\!+/ \restriction q,p \\
&= \{\text{ definition }\} \\
&\quad + / * \cdot n\varsigma \,\triangle\, \triangle \cdot n\copyright \restriction p \times q \cdot +\!\!\!+/ \restriction q,p \\
&= \{\text{ absorbing restriction, factorising } \copyright \} \\
&\quad + / * \cdot n\varsigma \,\triangle\, \triangle \cdot +\!\!\!+/ \cdot q\copyright \cdot p\copyright \cdot +\!\!\!+/ \restriction q,p \\
&= \{\text{ properties of } \copyright \text{ and } * \} \\
&\quad + / * \cdot n\varsigma \,\triangle\, \triangle \cdot +\!\!\!+/ \cdot +\!\!\!+/ ** \cdot q\copyright \cdot p\copyright \restriction q,p \\
&= \{\text{ earlier calculation }\} \\
&\quad + / * \cdot +\!\!\!+/ \cdot +\!\!\!+/ ** \cdot K \cdot q\copyright \cdot p\copyright \restriction q,p \\
&\quad \text{where } K = n\varsigma \,\square\, \triangle \,\square\, \triangle \cdot n\varsigma \,\triangle * \square\, \triangle \cdot n\varsigma \,\square\, \triangle\, \triangle * \cdot n\varsigma \,\triangle * \triangle * \\
&= \{\ f* \text{ is a homomorphism }\} \\
&\quad +\!\!\!+/ \cdot +/ ** \cdot +\!\!\!+/ ** \cdot K \cdot q\copyright \cdot p\copyright \restriction q,p \\
&= \{\ * \text{ distributes over composition, } + \text{ is associative }\} \\
&\quad +\!\!\!+/ \cdot (+/ \cdot + / *) ** \cdot K \cdot q\copyright \cdot p\copyright \restriction q,p
\end{aligned}$$

But then, because $q\copyright \cdot p\copyright \restriction q,p = q,p,p,q \upharpoonleft q\copyright \cdot p\copyright$ the instance of K is applied only to values of type q,p,p,q.

$$\begin{aligned}
K \restriction q,p,p,q \;&= \{\text{ homogeneity of } n\varsigma \text{ and pointwise operations }\} \\
&\quad (p \times q)\varsigma \,\square\, \triangle \,\square\, \triangle \restriction q,p,p,q \cdot (p \times q)\varsigma \,\triangle * \square\, \triangle \restriction q,p,p,q \,\cdot \\
&\quad (p \times q)\varsigma \,\square\, \triangle\, \triangle * \restriction q,p,p,q \cdot (p \times q)\varsigma \,\triangle * \triangle * \restriction q,p,p,q \\
&= \{\text{ properties of } \square \text{ and pointwise operators }\} \\
&\quad (p \times q)\varsigma^{p \times q} * \triangle * \triangle \restriction q,p,p,q \cdot (p \times q)\varsigma^{p} \,\triangle ** \triangle \restriction q,p,p,q \,\cdot \\
&\quad (p \times q)\varsigma^{q} * \triangle\, \triangle * \restriction q,p,p,q \cdot (p \times q)\varsigma \,\triangle * \triangle * \restriction q,p,p,q \\
&= \{\text{ properties of } \varsigma \} \\
&\quad 1\varsigma * \triangle * \triangle \restriction q,p,p,q \cdot q\varsigma \,\triangle ** \triangle \restriction q,p,p,q \,\cdot \\
&\quad p\varsigma * \triangle\, \triangle * \restriction q,p,p,q \cdot n\varsigma \,\triangle * \triangle * \restriction q,p,p,q \\
&= \{\ 1\varsigma \text{ is cancellable, and commuting terms }\} \\
&\quad *,*,p,q \upharpoonleft q\varsigma \,\triangle ** \triangle \cdot n\varsigma \,\triangle * \triangle * \cdot p\varsigma * \triangle\, \triangle * \restriction q,p,*,*
\end{aligned}$$

Substituting back into the main calculation

$$
\begin{aligned}
\mathcal{F} \upharpoonright n \cdot +\!\!+/ \upharpoonright q, p \;&=\; +\!\!+/ \cdot (+/ \cdot + / *) * * \upharpoonright *, *, p, q \;\cdot \\
&\qquad q\varsigma \triangle * * \triangle \cdot n\varsigma \triangle * \triangle * \cdot p\varsigma * \triangle \triangle * \;\cdot \\
&\qquad q, p, *, * \upharpoonleft q© \cdot p© \upharpoonright q, p \\
&=\; +\!\!+/ \upharpoonright p, q \cdot (+/ \cdot + / *) * * \;\cdot \\
&\qquad q\varsigma \triangle * * \triangle \cdot n\varsigma \triangle * \triangle * \cdot p\varsigma * \triangle \triangle * \;\cdot \\
&\qquad q© \cdot p© \upharpoonright q, p
\end{aligned}
$$

The strategy from this point is to use the equality $k© \cdot f = f* \cdot k©$, and the distributivity of ς over addition, that is $+/ \cdot k\varsigma* = k\varsigma \cdot +/$ to simplify by eliminating some of the $*$ operators from the expression. To do this the order of some of the operators must be changed by composing both sides with a transposition.

$$
\begin{aligned}
\mathcal{F} \upharpoonright n \cdot +\!\!+/ \upharpoonright q, p \cdot \text{И} \;&=\; +\!\!+/ \upharpoonright p, q \cdot (+/ \cdot + / *) * * \;\cdot \\
&\qquad q\varsigma \triangle * * \triangle \cdot n\varsigma \triangle * \triangle * \cdot p\varsigma * \triangle \triangle * \;\cdot \\
&\qquad q© \cdot p© \cdot \text{И} \upharpoonright p, q \\
&= \{\text{ transposing pointwise operations }\} \\
&\qquad +\!\!+/ \upharpoonright p, q \cdot (+/ \cdot + / *) * * \cdot \text{И} * * \;\cdot \\
&\qquad q\varsigma * \triangle * \triangle \cdot n\varsigma * \triangle \triangle * \cdot p\varsigma \triangle * \triangle * \;\cdot \\
&\qquad q© \cdot p© \upharpoonright p, q \\
&= \{\text{ commutativity of } + \} \\
&\qquad +\!\!+/ \upharpoonright p, q \cdot (+/ \cdot + / *) * * \;\cdot \\
&\qquad q\varsigma * \triangle * \triangle \cdot n\varsigma * \triangle \triangle * \cdot p\varsigma \triangle * \triangle * \;\cdot \\
&\qquad q© \cdot p© \upharpoonright p, q \\
&= \{\text{ distributivity of } \varsigma \text{ over } + \} \\
&\qquad +\!\!+/ \upharpoonright p, q \cdot +/ * * \cdot q\varsigma\triangle * \triangle \cdot n\varsigma\triangle \triangle * \;\cdot \\
&\qquad +/* * * \cdot p\varsigma \triangle * \triangle * \cdot q© \cdot p© \upharpoonright p, q \\
&= \{\text{ carrying } q© \text{ across } * \} \\
&\qquad +\!\!+/ \upharpoonright p, q \;\cdot \\
&\qquad +/ * * \cdot q\varsigma\triangle * \triangle \cdot q© \cdot n\varsigma \triangle \triangle \;\cdot \\
&\qquad +/ * * \cdot p\varsigma\triangle * \triangle \cdot p© \upharpoonright p, q \\
&= \{\text{ factorising and distributing restriction }\} \\
&\qquad +\!\!+/ \upharpoonright p, q \;\cdot \\
&\qquad +/ * * \cdot q\varsigma\triangle * \triangle \cdot q© \upharpoonright q, * \;\cdot \\
&\qquad n\varsigma \triangle \triangle \;\cdot \\
&\qquad +/ * * \cdot p\varsigma\triangle * \triangle \cdot p© \upharpoonright p, *
\end{aligned}
$$

There are two occurrences of similar expressions in the right-hand side, differing only in the parameter p or q. Each of these can be shown in the same way to satisfy

$$
\begin{aligned}
+/ * * \cdot k\varsigma\triangle * \triangle \cdot k© \upharpoonright k, * \;&= \{\, k© \upharpoonright *, * = \text{И} \cdot k © * \,\} \\
&\qquad +/ * * \cdot k\varsigma\triangle * \triangle \cdot \text{И} \cdot k © * \upharpoonright k, *
\end{aligned}
$$

$$
\begin{aligned}
&= \{\text{ transposition of pointwise operations }\}\\
&\qquad \text{И} \cdot +/ \ast\ast \cdot k\varsigma \triangle\, \triangle \ast \cdot k \circledc \ast \upharpoonright k, \ast\\
&= \{ \ast \text{ distributes over composition }\}\\
&\qquad \text{И} \cdot (+/\ast \cdot k\varsigma \triangle\, \triangle \cdot k \circledc \upharpoonright k) \ast\\
&= \{\text{ definition }\}\\
&\qquad \text{И} \cdot (\mathcal{F} \upharpoonright k)\ast
\end{aligned}
$$

so showing that

$$
\mathcal{F} \upharpoonright n \cdot +\!\!\!+/ \upharpoonright q, p \cdot \text{И} \quad = \quad +\!\!\!+/ \upharpoonright p, q \cdot \text{И} \cdot (\mathcal{F} \upharpoonright q)\ast \cdot n\varsigma \triangle\, \triangle \cdot \text{И} \cdot (\mathcal{F} \upharpoonright p)\ast
$$

Now И is its own inverse, and so can be carried over to the other side of the equation. Moreover, $+\!\!\!+/ \upharpoonright q, p$ is a bijection onto the set of $q \times p$-lists – which is anyway the domain of $\mathcal{F} \upharpoonright (p \times q)$ – and so can be inverted.

$$
\begin{aligned}
\mathcal{F} \upharpoonright n \quad &= \quad +\!\!\!+/ \upharpoonright p, q \cdot \text{И} \cdot (\mathcal{F} \upharpoonright q)\ast \cdot n\varsigma \triangle\, \triangle \cdot \text{И} \cdot (\mathcal{F} \upharpoonright p)\ast \cdot \text{И} \cdot (+\!\!\!+/ \upharpoonright q, p)^{-1}\\
&= \quad +\!\!\!+/ \upharpoonright p, q \cdot \text{И} \cdot (\mathcal{F} \upharpoonright q)\ast \cdot n\varsigma \triangle\, \triangle \cdot \text{И} \cdot (\mathcal{F} \upharpoonright p)\ast \cdot \text{И} \cdot q, p \upharpoonleft (+\!\!\!+/)^{-1}
\end{aligned}
$$

Allowing for a slight abuse of notation, the bizarre looking function $q, p \upharpoonleft (+\!\!\!+/)^{-1}$ is that which takes a list of length n and divides it into p chunks each of length q. The remaining asymmetry in the expression is annoying, but merely superficial for of course $n\varsigma \triangle\, \triangle \cdot \text{И} = \text{И} \cdot n\varsigma \triangle\, \triangle$.

The decomposition of $\mathcal{F} \upharpoonright n$ can be read – taking terms from right to left – as a divide-and-conquer algorithm for implementing transforms of composite width: divide the input into p chunks of length q; interleave them; apply an array (of q) independent p-point transforms; interleave the results; modify by scaling the $\langle i, j \rangle$-th signal by $(n\varsigma)^{i \times j}$; apply an array (of p) independent q-point transforms; interleave the results; and finally concatenate the q resulting lists, each of which is of length p, into a single n-list. This is the algorithm known as the 'fast Fourier transform'.

It is the contention of this paper that this equation shows much more clearly than the manipulation of summations that a discrete Fourier transform can be implemented by a divide-and-conquer algorithm using a number of smaller transforms of the same kind.

Twiddling

Leaving aside the rearrangement of the data, on unwinding the recursive calls it transpires that all the substantial work performed by the fast Fourier transform algorithm is in the application of $n\varsigma \triangle\, \triangle \upharpoonright p, q$.

Suppose that $\oplus$ and $\otimes$ are operators that *cross-associate*, in the sense that $(x \oplus y) \otimes z \;=\; x \oplus (y \otimes z)$, and that $\otimes$ has a left unit $\iota_\otimes$, then it can easily be shown by induction on k that $(x\oplus)^k = ((x\oplus)^k \iota_\otimes)\otimes$. Since each component of $\iota_{\Upsilon_\otimes}$ is necessarily $\iota_\otimes$, it follows immediately that $x \oplus \triangle = (x \oplus \triangle\, \iota_{\Upsilon_\otimes})\Upsilon_\otimes$.

Again, since the associativity of $\Upsilon_\otimes$ follows from that of $\otimes$, and since any associative operator cross-associates with itself,

$$
\begin{aligned}
x\oplus \triangle\, \triangle \quad &= \{\text{ applying the lemma to } \oplus \text{ and the inner } \triangle \}\\
&\qquad (x\oplus \triangle\, \iota_{\Upsilon_\otimes})\, \Upsilon_\otimes \triangle
\end{aligned}
$$

$= \{$ applying the lemma to $\Upsilon_\otimes$ and the outer $\triangle$ $\}$

$((x\oplus \triangle\ \iota_{\Upsilon_\otimes})\ \Upsilon_\otimes \triangle \iota_{\Upsilon_{\Upsilon_\otimes}})\ \Upsilon_{\Upsilon_\otimes}$

$= \{$ inverting the lemma, for $\oplus$ and the inner $\triangle$ $\}$

$(x \oplus \triangle \triangle\ \iota_{\Upsilon_{\Upsilon_\otimes}})\Upsilon_{\Upsilon_\otimes}$

This decomposition gives an algorithm for calculating $x\oplus \triangle \triangle$ by an $O(p{\times}q) = O(n)$ linear-time application of $\Upsilon_{\Upsilon_\otimes}$ to the signal and a term which depends only on the size of the circuit.

In the case of the Fourier transform, this term is an array of elements of the underlying integral domain – frequently referred to mysteriously as the 'twiddle factors' – which can therefore be pre-calculated. The $\otimes$ operation corresponding to ς is the multiplication on the integral domain, and its left unit is the unit of the domain, so

$$n\varsigma \triangle \triangle \restriction p, q \quad = \quad (n\ \varsigma \triangle \triangle \mathbf{1})\Upsilon_{\Upsilon_\times}$$

where $\mathbf{1}$ is the appropriately-sized (two-dimensional) array of ones. The $n\ \varsigma \triangle \triangle \mathbf{1}$ term is the array of twiddle factors, and they can be applied by about $p \times q$ multipliers arranged according to $\Upsilon_{\Upsilon_\times}$. To be precise, only $(p{-}1){\times}(q{-}1)$ multipliers can be needed since $p + q - 1$ of the twiddle factors are guaranteed to be one.

Outline of an implementation

The usual recursive 'butterfly' implementation of the fast Fourier transform applies only to transforms on vectors of length 2^n for some n. This is because it is very easy to do two-point transforms: because minus one is the principal square root of unity, the two-point transform $\Phi = \mathcal{F} \restriction 2$ takes $\langle x_0, x_1\rangle$ into $\langle x_0 + x_1, x_0 - x_1\rangle$ and requires no multiplications.

For higher powers of two, it uses the factorisation

$$\mathcal{F} \restriction 2n \quad = \quad \mathbin{+\!\!+}/ \restriction 2, n \cdot \textbf{И} \cdot (\mathcal{F} \restriction n)* \cdot (2n\ \varsigma \triangle \triangle \mathbf{1})\Upsilon_{\Upsilon_\times} \cdot \textbf{И} \cdot \Phi* \cdot \textbf{И} \cdot n, 2 \upharpoonleft (\mathbin{+\!\!+}/)^{-1}$$

The function $n, 2 \upharpoonleft (\mathbin{+\!\!+}/)^{-1}$ divides its input into halves of length n, and $\mathbin{+\!\!+}/ \restriction 2, n$ joins a list of n pairs. The only explicit multiplications in this factorisation are in the $\Upsilon_{\Upsilon_\times}$ operator, and can be implemented by an array of $2n$ multiplications only $n-1$ of which are non-trivial. The factorisation is used recursively on the $\mathcal{F} \restriction n$ term until only two-point transforms remain.

The usual way of implementing this algorithm – that is, the usual way of laying out the circuit – is to divide $\mathcal{F}$ into two parts: let $\mathcal{F} = \mathcal{S} \cdot \mathcal{F}'$ where

$$\mathcal{F}' \restriction 2n \quad = \quad \mathbin{+\!\!+}/ \restriction n, 2 \cdot (\mathcal{F}' \restriction n)* \cdot (2n\ \varsigma \triangle \triangle \mathbf{1})\Upsilon_{\Upsilon_\times} \cdot \textbf{И} \cdot \Phi* \cdot \textbf{И} \cdot n, 2 \upharpoonleft (\mathbin{+\!\!+}/)^{-1}$$

(A transposition has been omitted from the left-hand end of the expression, which explains the type of the final catenation.) Sheeran [17, 18] gives a solution, usually known as a 'butterfly' circuit, to the recursion

$$\mathcal{B} \restriction 2n \quad = \quad \mathbin{+\!\!+}/ \restriction n, 2 \cdot (\mathcal{B} \restriction n)* \cdot \textbf{И} \cdot \Phi* \cdot \textbf{И} \cdot n, 2 \upharpoonleft (\mathbin{+\!\!+}/)^{-1}$$

which is almost the same as that for $\mathcal{F}'$. The solution need only be adjusted – perhaps using Luk's heterogeneous constructors [19] – to accommodate the twiddle factors. Alternately, the twiddle factors might be calculated in a pre-pass using another co-located butterfly of the same shape as $\mathcal{B}$.

Similarly, the solution to

$$\mathcal{S} \upharpoonright 2n \;\; = \;\; +\!\!+/ \upharpoonright 2, n \cdot \text{И} \cdot (\mathcal{S} \upharpoonright n)* \cdot n, 2 \upharpoonleft +\!\!+/$$

is a permutation. It is that very thorough shuffle which appears inscrutably in implementations of the fast transform, and which reverses the bits of the index of the position of a value in a vector.

It should also be clear from this paper that the butterfly implementation can be extended to input of any width n, given only implementations of $\mathcal{F} \upharpoonright p$ for each prime p which divides n. This generalization is also suggested by Cooley and Tukey.

Estimation of costs

The reason that the 'fast' algorithm is so called is that it requires a significant number fewer of the basic arithmetic operations. As shown, you can implement $n\varsigma \bigtriangleup \bigtriangleup \upharpoonright p, q$ for a given n by $(p-1) \times (q-1)$ circuits which multiply by constants, and $+/* \upharpoonright p, q$ by $(p-1) \times q$ adders. Accordingly, the algorithm suggested by

$$\mathcal{F} \upharpoonright n \;\; = \;\; +/* \cdot n\varsigma \bigtriangleup \bigtriangleup \cdot n\text{©} \upharpoonright n$$

seems to require $(n-1)^2$ multipliers and $n \times (n-1)$ adders.

Suppose the fast algorithm requires at most M_k multipliers and A_k adders to implement $\mathcal{F}_k$, for $k < n$, then if n is composite, then for any factorisation into p and q

$$\mathcal{F} \upharpoonright n \;\; = \;\; +\!\!+/ \upharpoonright p, q \cdot \text{И} \cdot (\mathcal{F} \upharpoonright q)* \cdot n\varsigma \bigtriangleup \bigtriangleup \cdot \text{И} \cdot (\mathcal{F} \upharpoonright p)* \cdot \text{И} \cdot q, p \upharpoonleft (+\!\!+/)^{-1}$$

gives an algorithm which shows that

$$\begin{aligned} M_n &\leqslant p \times M_q + (p-1) \times (q-1) + q \times M_p \\ A_n &\leqslant p \times A_q + q \times A_p \end{aligned}$$

Each of these gives a bound of $O(n \log n)$ on the number of active components.

In the specific case of the power-of-two sized transform, the number of multiplications (discounting mere sign-changes) can be shown to be $(n \log n)/(2 \log 2) - n + 1$, and the number of additions (or subtractions) is $n \log n$.

Transforms over vectors

Having set up the mechanism, it is worth showing that we can also explain within this formalism the sense in which the fast transform is related to transforms on vectors. Let $\mathcal{G}$ be the discrete Fourier transform, abstracted on the arithmetic operations:

$$\mathcal{G}(\oplus, \otimes) \upharpoonright n \;\; = \;\; \oplus/* \cdot n\otimes \bigtriangleup \bigtriangleup \cdot n\text{©} \upharpoonright n$$

A term in the factorisation of a discrete Fourier transform has the form

$$
\begin{aligned}
\textbf{И} \cdot (\mathcal{G}(\oplus, \otimes) \upharpoonright n)* \cdot \textbf{И} \quad &= \{\, \text{definition} \,\} \\
&\quad \textbf{И} \cdot (\oplus/* \cdot n\otimes \,\triangle\, \triangle \cdot n\copyright \upharpoonright n)* \cdot \textbf{И} \\
&= \{\, \text{distribution of } * \text{ over composition} \,\} \\
&\quad \textbf{И} \cdot \oplus/ ** \cdot n\otimes \,\triangle\, \triangle* \cdot n \copyright * \upharpoonright n, * \cdot \textbf{И} \\
&= \{\, \text{transposition exchanges pointwise operators} \,\} \\
&\quad \oplus/ ** \cdot n\otimes \,\triangle\, *\triangle \cdot \textbf{И} \cdot n \copyright * \upharpoonright n, * \cdot \textbf{И} \\
&= \{\, \text{properties of transpose and } \copyright \,\} \\
&\quad \oplus/ ** \cdot n \otimes \triangle * \triangle \cdot n\copyright \upharpoonright n, * \cdot \textbf{И} \\
&= \{\, \text{properties of transpose and } \copyright \,\} \\
&\quad \oplus/ ** \cdot n \otimes \triangle * \triangle \cdot \textbf{И}* \cdot n\copyright \upharpoonright *, n \\
&= \{\, \text{transposition exchanges pointwise operators} \,\} \\
&\quad \oplus/ ** \cdot \textbf{И}* \cdot n \otimes * \triangle \triangle \cdot n\copyright \upharpoonright n \\
&= \{\, \text{transposition rule for } \Upsilon_{\oplus}/ \,\} \\
&\quad \Upsilon_{\oplus}/* \cdot (n \otimes *) \triangle \triangle \cdot n\copyright \upharpoonright n \\
&= \{\, \text{definition} \,\} \\
&\quad \mathcal{G}(\Upsilon_{\oplus}, (*) \cdot (\otimes)) \upharpoonright n
\end{aligned}
$$

but if $n\otimes$ is a scaling of a value of some type, then $((*) \cdot (\otimes))n = (*)((\otimes)n) = (n\otimes)*$ is the corresponding scaling of vectors of that type; and if $\oplus$ is addition of values of some type, then $\Upsilon_{\oplus}$ is addition of vectors of that type.

This has related a number of scalar transforms on interleaved samples to a transform on vectors of samples. You can therefore, if you so wish, see the term $\textbf{И} \cdot (\mathcal{F} \upharpoonright n)* \cdot \textbf{И}$ in the factorisation of the transform as an n-point vector-valued transform operating on a list of n vectors.

Summary

A notation has been presented for describing circuits, and a framework for reasoning about them. This framework was previously known to be able to deal with simple, regular circuits of the sort that have little wiring in their layout. Having suggested that these techniques were suitable for designing digital signal-processing circuits, it was necessary to show that they could deal with existing signal processing components.

The specification of the discrete Fourier transform was translated into the notation; and a summary was given of a calculation from that specification of the fast Fourier transform algorithm. The details of the calculation appear in another paper [20], where it is conducted at a level that could be explained to a very simple mechanical proof checker. Reference [7] contains essentially the same summary of the calculation as is in the present paper, but in a different notation. The presentation given here seems to be a little more natural, at least to users of the Bird-Meertens formalism.

Because all of the reasoning is based on the algebra of the operators, and not on the specific operators or type of data, the calculation is independent of the choice of that type. Although the calculation might have seemed to be about a circuit that would transform a constant input vector, earlier work [21] has shown that the algebra is unchanged by a systematic re-interpretation of all the operators as ones that operate, for example, on time-sequences. (This re-interpretation is known elsewhere as as 'lifting' [22].) It follows that our development of the fast algorithm applies equally well to a circuit that would operate on a time-sequence of sample vectors.

Although the detailed calculation – like any calculation carried out in great detail – is difficult to follow, we contend that

$$\begin{aligned} &\mathcal{F} \upharpoonright (p \times q) \\ &\quad = \quad +\!\!\!+/ \upharpoonright p, q \cdot \text{И} \cdot (\mathcal{F} \upharpoonright q)* \cdot ((p \times q) \varsigma \triangle \triangle \mathbf{1}) \Upsilon_{\Upsilon_\times} \cdot \text{И} \cdot (\mathcal{F} \upharpoonright p)* \cdot \text{И} \cdot q, p \upharpoonleft (+\!\!\!+/)^{-1} \end{aligned}$$

shows much more clearly than any expression describing the output (rather than the circuit) that a discrete Fourier transform can be implemented by a divide-and-conquer strategy which yields a number of smaller transforms of the same kind.

The derivation of this equation proved also to be necessary in writing a program to implement the fast Fourier transform: without the equation it would be hard to understand the program; without the derivation it would be hard to believe the equation. The relevant fragment of the program is

```
fft (+/) ($igma) = f
                        where f   xs = f' xs,  if prime (#xs)
                                     = f'' xs, otherwise
                                f'   = dft (+/) ($igma)
                                f'' xs = ((++/) . (|/|) . (f *) .
                                           (((n $igma) /\) /\) .
                                          (|/|) . (f *) . (|/|) . (\++)) xs
                                         where n      = #xs
                                               (\++) = into (factor n)
```

A complete Orwell [23] program demonstrating the execution of this function appears as an appendix to this paper.

Performing the detailed derivation of fast Fourier transform from the specification has shown that it is reasonable to expect to be able to perform this derivation using the sort of machine assistance available for supporting array designs in the same style. It might also be possible to extend this to mechanical layout of high-wire circuits like the butterfly implementation of the fast Fourier transform, including allocation of the correct twiddle factor to the correct multiplier. Although this necessarily detailed study of the fast Fourier algorithm has revealed structure which was not previously so clear, the principal outcome of the work is to support our claim that our functional circuit-design style can indeed deal with another kind of digital signal processing circuit.

I am grateful to Mary Sheeran, for having spared me the necessity of the other half of this work; Lambert Meertens, for some unfortunate notational advice which I have nevertheless taken to my heart; and to the squiggolists at the Programming Research Group, for tolerating this calculation in its unruly adolescence.

References

[1] Sheeran M. *Design and verification of regular synchronous circuits*, IEE Proceedings, vol. 133, Pt. E, No. 5, September 1986, pp 295–304

[2] Bird RS. *An introduction to the theory of lists*, In: Broy M (ed) *Logic of programming and calculi of discrete design*, Springer, 1987, pp 3–42

[3] Bird RS. *Lectures on constructive functional programming*, Oxford University Computing Laboratory Programming Research Group technical monograph PRG–69, September 1988

[4] Luk W, Jones G. *The derivation of regular synchronous circuits*, In: Bromley K, Kung S-Y, Schwarzlander E. *Proceedings of the international conference on systolic arrays*, IEEE Computer Society Press, 1988, pp 305–314

[5] Luk W, Jones G. *From specification to parametrised architectures*, In: Milne GJ (ed) *The fusion of hardware design and verification*, North-Holland, 1988, pp 267–288

[6] Sheeran M, Jones G. *Relations + higher-order functions = hardware descriptions*, Proc. CompEuro 89, Hamburg, May 1987 pp 303–306

[7] Jones G. *Calculating the fast Fourier transform as a divide and conquer algorithm*, (unpublished)

[8] Jones G. *Factorising Fourier for fastness*, (privately circulated draft of present paper)

[9] Jones G. *Constructing the fast Fourier transform by calculating with the algorithm*, working paper for the STOP summer school on *Constructive Algorithmics*, Ameland, September 1989

[10] Cooley JW, Tukey JW. *An algorithm for the machine computation of complex Fourier series*, Mathematics of Computation, **19**, 1965, pp 297–301

[11] Aho AV, Hopcroft JE, Ullman JD. *The design and analysis of computer algorithms*, Addison–Wesley, 1974

[12] Smith SG. *Fourier transform machines*, In: Denyer P, Renshaw D. *VLSI signal processing; a bit-serial approach*, Addison–Wesley, 1985, pp 147–199

[13] Ullman JD. *Computational aspects of VLSI*, Computer Science Press, 1984

[14] Chandy KM, Misra J. *Parallel program design – a foundation*, Addison–Wesley, 1988

[15] Jones G, Luk W. *Exploring designs by circuit transformation*, In: Moore W, McCabe A, Urquhart R (eds) Proceedings of the first international workshop on *Systolic Arrays*, Adam Hilger, 1987, pp 91–98

[16] Sheeran M. *Retiming and slowdown in Ruby*, In: Milne GJ (ed) *The fusion of hardware design and verification*, North-Holland, 1988, pp 289–308

[17] Sheeran M. *Describing hardware algorithms in Ruby*, In: David et al (eds) Proceedings IFIP WG10.1 workshop on *Concepts and Characteristics of Declarative Systems*, Budapest, 1988

[18] Sheeran M, Jones G. *Butterfly algorithms*, (in hand)

[19] Luk W. *Parametrised design for regular processor arrays*, D.Phil. thesis, Oxford University, 1988

[20] Jones G. *Fast Fourier transform by program transformation of the discrete Fourier transform*, (submitted for publication)

[21] Jones G, Sheeran M. *Timeless truths about sequential circuits*, In: Tewksbury SK, Dickinson BW, Schwartz SC (eds) *Concurrent computations: algorithms, architectures and technology*, Plenum Press, 1988, pp 245–259

[22] Backhouse RC. *Making formality work for us*, Technical Report CS 8907, Department of Mathematics and Computing Science, University of Groningen, 1989

[23] Wadler PL, Miller Q. *An introduction to Orwell 5.00*, Oxford University Programming Research Group, 1988

An Orwell script for fast Fourier transform

```
This script contains implementations of the discrete Fourier transform,
        dft -- an quadratic implementation close to the specification
        fft -- an O(n.logn) recursive implementation
        vft -- a sketch of an optimisation of the same
as described in the accompanying paper.  There are also code to exercise
them and examples of its use.
                                                            gj 3.ix.1989
Operator declarations
~~~~~~~~~~~~~~~~~~~~~
The odd graphics which are used as operators have to be introduced at the
top of the script.

Although * is used thoughout this script for ‘map’, it has to be bound
in local definitions because its use has been pre-empted by predefined
numeric multiplication.  Otherwise it would have been defined here by

  %right   90  *
  (*) :: (a -> b) -> [a] -> [b]
  (*) = map

The triangle operator satisfies (f /\ x)!i = (f^i) (x!i), but is defined
recursively

> %right   90  /\
> (/\) :: (a -> a) -> [a] -> [a]
> f /\    []  = []
> f /\ (x:xs) = x : (f /\ (f * xs))
>                where (*) = map

Transposition satisfies (|/| xss)!i!j = xss!j!i
```

```
> %prefix  |/|
> (|/|) :: [[a]] -> [[a]]
> |/| xss = [ [x]   |  x     <-      hd xss                ], if #xss = 1
>         = [  x:xs | (x,xs) <- zip (hd xss, |/| (tl xss)) ], if #xss > 1
```

The operator +/ will be used to stand for a various addition-reductions, and although no value is given here, it needs to be declared to be a prefix operator

```
> %prefix +/
```

List catenation and its inverse

Catenation of lists, written ++/ in the paper, is implemented by the primitive concat = foldr (++) [] for speed

```
> %prefix ++/
> (++/) :: [[a]] -> [a]
> (++/) = concat
```

It is inverted by two functions, ‘by’ and ‘into’: (by n xs) is used only if #xs is divisible by n, and returns a list of lists of length n which when concatenated return xs

```
> by :: num -> [a] -> [[a]]
> by n xs = [],                           if xs = []
>         = take n xs : by n (drop n xs), if #xs >= n
```

and (into n xs) is used only if #xs is divisible by n, and returns a list of length n of lists which when concatenated return xs

```
> into :: num -> [a] -> [[a]]
> into n xs = by (#xs $div n) xs
```

Instances of by and into will be represented by the \++ operator (which is written as the inverse of ++/ in the paper) here introduced by

```
> %prefix \++
```

Fourier transform

The Fourier transform of xs is obtained by summing xs_i.w^(i.j); the transform is implemented by (dft (+/) ($igma)) where (n $igma) is a function that represents multiplication by the n-th root of unity, and ((+/) xs) represents the sum of the elements of xs.

```
> dft :: ([a] -> a) -> (num -> a -> a) -> ([a] -> [a])
> dft (+/) ($igma) xs = (((+/) *) . (((n $igma) /\) /\) . (n $copy)) xs
>                       where n   = #xs
>                             (*) = map
```

and the fast transform is obtained by a divide-and-conquer strategy applied to composite width vectors, falling back on the use of dft in the other cases

```
> fft :: ([a] -> a) -> (num -> a -> a) -> ([a] -> [a])
> fft (+/) ($igma) = f
>                   where f   xs = f' xs,  if prime (#xs)
>                                = f'' xs, otherwise
>                         f'     = dft (+/) ($igma)
>                         f'' xs = ((++/) . (|/|) . (f *) .
>                                    (((n $igma) /\) /\) .
>                                   (|/|) . (f *) . (|/|) . (\++)) xs
>                                  where n     = #xs
>                                        (\++) = into (factor n)
>                                        (*)   = map
```

Here, the factor of n chosen for division of xs is (factor n) which is the largest factor of n less than its square root, that is

```
> factor n = (last . divides n . belowroot n) [1..n]
>              where divides n   = filter ((= 0) . (n $mod))
>                    belowroot n = takewhile ((<= n) . sq)
>                                  where sq x = x * x
```

and n is prime iff this factor is one.

```
> prime n = factor n = 1
```

(I know it could be made more efficient, but I was writing for clarity!)

You might try to replace the term ((|/|) . (f *) . (|/|)), according to the paper, by (fft (((+/) *) . (|/|)) ((*) . ($igma))), and indeed

```
  vft :: ([a] -> a) -> (num -> a -> a) -> ([a] -> [a])
  vft (+/) ($igma) = f
                    where f   xs = f' xs,  if prime (#xs)
                                 = f'' xs, otherwise
                          f'     = dft (+/) ($igma)
                          f'' xs = ((++/) . (|/|) . (f *) .
                                     (((n $igma) /\) /\) .
                                              f''' . (\++)) xs
                                   where n     = #xs
                                         (\++) = into (factor n)
                                         (*)   = map
                          f'''   = vft vsum vsigma
                                   where vsum   = ((+/) *) . (|/|)
                                                  where (*) = map
                                         vsigma = (*) . ($igma)
                                                  where (*) = map
```

would work were it not for the fact that the recursive call leads to an ill-founded type recursion: the arguments of the recursive call of vft oblige it to have a type

```
        ([[a]] -> [a]) -> (num -> [a] -> [a]) -> ([[a]] -> [[a]])
```

You can make a definition which uses this algorithm for one factorisation and then falls back on the earlier algorithm:

```
> vft :: ([a] -> a) -> (num -> a -> a) -> ([a] -> [a])
> vft (+/) ($igma) = f
>                where f   xs = f' xs,  if prime (#xs)
>                           = f'' xs, otherwise
>                      f'     = dft (+/) ($igma)
>                      f'' xs = ((++/) . (|/|) . (f *) .
>                                  (((n $igma) /\) /\) .
>                                           f''' . (\++)) xs
>                               where n     = #xs
>                                     (\++) = into (factor n)
>                                     (*)   = map
>                      f'''   = fft vsum vsigma
>                               where vsum   = ((+/) *) . (|/|)
>                                              where (*) = map
>                                     vsigma = (*) . ($igma)
>                                              where (*) = map
```

The large number of bindings of (*) are again an artefact of the type checking, since the uses have different types.

A symbolic example

The arithmetic in the example is going to be over formal polynomials in a given root of unity, that root being the smallest one to be used in a given program. The polynomial -- necessarily of finite degree -- is represented by a list of its coefficients, in decreasing order of power of the root.

```
> poly a == [a]
```

Multiplication of such a polynomial by the given root of unity moves the coefficient of the highest power of the root to the end of the list -- as the coefficient of unity in the result -- and shifts the other coefficients left by one place.

```
> shift :: [a] -> [a]
> shift (x:xs) = xs ++ [x]
```

If p divides #xs, then (p $sigma xs) represents xs times the p-th principal root of unity: this is achieved by shifting the coefficients by one p-th of their length.

```
> ($igma) :: num -> poly a -> poly a
> (p $igma) = (++/) . shift . into p
```

The sum of two formal polynomial values is the pointwise sum of the coefficients; since each of these coefficients will be represented by a list of terms, they can just be concatenated to represent addition. The sum of a list of polynomials can either be implemented by a fold of a zipwith, or more conveniently for the present purposes by the equivalent

```
> (+/) :: [poly a] -> poly a
> (+/) = ((++/) *) . (|/|) where (*) = map
```

Demonstration code
~~~~~~~~~~~~~~~~~~

To demonstrate the symbolic Fourier transform code, it will be exercised on polynomials, of degree less than n, over lists of integers; the presence of the number i in the j-th coefficient of a polynomial (indexed of course from the right-hand end) will be taken to represent a formal variable x_i times the j-th power of w -- a formal n-th root of unity.

```
> index == num
> coeff == [index]
```

A vector of length n, of formal polynomials of degree less than n, will be laid out by translating each of the polynomials into strings and laying them out one to a line.

```
> showvec :: [poly coeff] -> string
> showvec = layn . map showpoly
```

Each polynomial is represented by displaying the terms with non-zero (non-empty) coefficients, and punctuating with '+' signs.

```
> showpoly :: poly coeff -> string
> showpoly = stringfold " + " . map showterm . index
>            where index xs
>                    = [ (x,i) | (x,i) <- zip(reverse xs,[0..]); x ~= []]
```

Each term is represented by the coefficient times 'w' to a power.

```
> showterm :: (coeff, num) -> string
> showterm (x,i) = showcoeff x,                            if i = 0
>                = bracket showcoeff x ++ ".w^" ++ show i, if i > 0
>                  where bracket f x = f x,              if #x = 1
>                                    = "(" ++ f x ++ ")", if #x > 1
```

Each coefficient is represented as a list of variables punctuated with '+' signs. The call of 'sort' arranges that the variables appear in
~~~~~~~~~~~~~~~~~~

order of index -- it happens that no variable will appear twice in any coefficient, but in principle this function could collect the multiplicity of a variable and pass it to ‘showvar’.

```
> showcoeff :: coeff -> string
> showcoeff = stringfold "+" . map showvar . sort
```

Each variable is represented by ‘x’ followed by its index.

```
> showvar :: index -> string
> showvar i = "x" ++ show i
```

The function ‘stringfold’ inserts the representation of an operator in the places in the representation of an expression corresponding to a reduction of that operator.

```
> stringfold :: string -> [string] -> string
> stringfold op = foldr1 (infix op)
>                 where infix op l r = l ++ op ++ r
```

The input to which an n-point transform will be applied is to represent a list of n polynomials, the i-th of which represents x_i (as the coefficient of unity in a formal polynomial of degree n-1).

```
> input :: num -> [poly num]
> input n = [ nulls ++ [[i]] | i <- [1..n] ] where nulls = copy (n-1) []
```

Execution of the example code
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

For example, the input for an eight-point transform is demonstrated by

```
? (showvec . input) 8

   1) x1
   2) x2
   3) x3
   4) x4
   5) x5
   6) x6
   7) x7
   8) x8
```

The specification applied to this input returns

```
? (showvec . dft (+/) ($igma) . input) 8

   1) x1+x2+x3+x4+x5+x6+x7+x8
   2) x1 + x2.w^1 + x3.w^2 + x4.w^3 + x5.w^4 + x6.w^5 + x7.w^6 + x8.w^7
   3) x1+x5 + (x2+x6).w^2 + (x3+x7).w^4 + (x4+x8).w^6
   4) x1 + x4.w^1 + x7.w^2 + x2.w^3 + x5.w^4 + x8.w^5 + x3.w^6 + x6.w^7
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
   5) x1+x3+x5+x7 + (x2+x4+x6+x8).w^4
   6) x1 + x6.w^1 + x3.w^2 + x8.w^3 + x5.w^4 + x2.w^5 + x7.w^6 + x4.w^7
   7) x1+x5 + (x4+x8).w^2 + (x3+x7).w^4 + (x2+x6).w^6
   8) x1 + x8.w^1 + x7.w^2 + x6.w^3 + x5.w^4 + x4.w^5 + x3.w^6 + x2.w^7

as do the similar expressions in fft and vft.

For what it is worth, on a Sun3 the costs are approximately:
 (showvec.dft(+/)($igma).input) 8    66 CPU seconds, 166 000 reductions
 (showvec.fft(+/)($igma).input) 8     4.8 seconds,     9 600 reductions
 (showvec.vft(+/)($igma).input) 8     5.0 seconds,     9 400 reductions

NB: these costs will not scale as O(n^2) and O(n.log n) because direct
    implementation of ((w /\) /\) requires O(n^4) applications of 'w'
    each of which might cost O(n).  In a practical implementation it
    would be necessary to do something more sensible.
```