The Fast Fourier Transform as a Database Query

Peter Buneman
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, Pa 19104, USA

The study of the common properties of programs that operate over lists, multisets, and sets, generically called *collection types*, has recently received some attention in database and programming language research [8, 2]. It has been especially important in extending the standard query language for relational databases — essentially first-order logic — to deal with the wider range of data types that are to be found, for example, in object-oriented databases. While the commonality between lists, multisets and sets is reasonably well understood, it is an open question as to whether arrays can be be successfully added to this family. The purpose of this note is to provide some positive evidence for this by showing that the fast Fourier transform can be derived from its specification, discrete Fourier transform, by casting it in the form of a database query and using an extension of the syntactic manipulation techniques already known for database query optimization.

That the FFT may be obtained by program transformation has already been demonstrated by Geraint Jones [5] using techniques based on the Bird-Meertens formalisms for lists [1]. However, derivation presented here is somewhat shorter and is, perhaps, more in tune with the mathematical syntax whose manipulation suggests that an $O(n \log n)$ algorithm may be obtained.

Comprehension syntax is based on Zermelo-Fraenkel set notation. For example $\{\text{Name } x \mid x \leftarrow \text{Employee}, \text{Age } x \geq 35\}$ and $\{(x,y) \mid (x,z) \leftarrow R, (t,y) \leftarrow S, z = t\}$ — the latter denoting the composition of binary relations R and S — are queries in relational calculus, which operates on sets of tuples. However they can also be viewed as programs on lists and are meaningful in certain functional programming languages [6]. It is known that such expressions can be transformed into an equivalent algebra of functions based on the categorical notion of a *monad* [7, 2] in which two central operations are a *mapping* operator $\vec{f}$ that applies f to each member of a collection and a *flattening* function μ that flattens a collection of collections into a single collection. For sets μ is "big" union; for lists it is concatenation. The behavior of these operations is described by

$$\vec{f}\{x_1, x_2, \ldots, x_n\} = \{f(x_1), f(x_2), \ldots, f(x_n)\} \tag{1}$$
$$\mu\{\{x_{1,1}, \ldots, x_{1,m_1}\}, \ldots, \{x_{n,1}, \ldots, x_{n,m_n}\}\} = \{x_{1,1}, \ldots, x_{m_1,1}, \ldots, x_{n,1}, \ldots, x_{n,m_n}\} \tag{2}$$

It helps to keep in mind the types of these operations. If τ is some type, let us use $\mathcal{C}(\tau)$ to describe the type of a collection of τ. If f is a function of type $\sigma \rightarrow \tau$, $\vec{f}$ is of type $\mathcal{C}(\sigma) \rightarrow \mathcal{C}(\tau)$; and for for each type τ, μ is a function of type $\mathcal{C}(\mathcal{C}(\tau)) \rightarrow \mathcal{C}(\tau)$

These operations are connected by an identity. If S has type $\mathcal{C}(\mathcal{C}(\mathcal{C}(\tau)))$,

$$\mu(\mu(S)) = \mu(\vec{\mu}(S)) \tag{3}$$

To see the relationship with comprehension syntax, observe that the general form of a comprehension is $\{e \mid c_1, \ldots, c_n\}$, where c_i is either a boolean expression or is a *generator* of the form $x_i \leftarrow e_i$ that introduces the variable x_i and binds it to each member of the collection denoted by the expression e_i. Once a variable has been introduced by a generator, it may be used in the "head" of the comprehension e or in subsequent components $c_{i+1}, \ldots, c_n$. In what follows we shall not be concerned with boolean components of comprehensions.

Using the mapping and flattening operations defined above, $\{e \mid x_i \leftarrow e_i, c_{i+1}, \ldots, c_n\}$ can be expressed as $\mu\{\{e \mid c_{i+1}, \ldots, c_n\} \mid x \leftarrow e'\}$; therefore we need only to deal with one-component comprehensions of the form $\{e \mid x \leftarrow e'\}$. Such comprehensions may be eliminated with the identity:

$$\{e \mid x \leftarrow e'\} \quad = \quad \vec{f}(e') \quad \text{where} \quad f(x) = e \tag{4}$$

Repeated applications of (4) can be used to eliminate completely comprehension syntax from an expression. Also from (4), the following identities may be derived:

$$\{f(e) \mid x \leftarrow e'\} = \quad \vec{f}\{e \mid x \leftarrow e'\} \quad = \{f(y) \mid y \leftarrow \{e \mid x \leftarrow e'\}\} \qquad \text{if } x \text{ does not occur in } f \tag{5}$$

If f is a function of type $\mathcal{C}(\tau) \rightarrow \tau$, we may have an identity related to 3 that $f(\mu(S)) = f(\vec{f}(S))$. For example, summation, Σ satisfies

$$\Sigma(\mu(S)) \quad = \quad \Sigma(\vec{\Sigma}(S)) \tag{6}$$

on lists or multisets of numbers, but *not* on sets.

A natural question to ask is whether arrays fit into this scheme of collections. For consistency, we must adopt the convention of many programming languages that arrays are one-dimensional. Mapping and flattening make sense for arrays, but we must now keep the length of arrays as part of their type. Using $\mathcal{A}^n(\tau)$ for the type of arrays of length n, a two dimensional $m \times n$ array has type $\mathcal{A}^m(\mathcal{A}^n(\tau))$. Thus for any m, n, τ, there is a flattening operation μ at type $\mathcal{A}^m(\mathcal{A}^n(\tau)) \rightarrow \mathcal{A}^{mn}(\tau)$. The equation 3 still holds for arrays. Each side has type $\mathcal{A}^l(\mathcal{A}^m(\mathcal{A}^n(\tau))) \rightarrow \mathcal{A}^{lmn}(\tau)$ but the intermediate types resulting from the applications of μ and $\vec{\mu}$ are now different.

Comprehension syntax also appears to make sense for arrays: $\{2x \mid x_i \leftarrow V\}$ will result in array whose ith component is double the ith component of V, and $\{ix \mid x_i \leftarrow V\}$ will result in an array whose ith component is i times the ith component of V. Note the deviation from mathematical convention, in which one would write $\{ix_i \mid x_i \leftarrow V\}$. In comprehension notation, the binding $x_i \leftarrow V$ is taken to bind both the variable x and the variable i to each successive value–index pair in V. It is also apparent that the identities 4, 5 and 6 hold for arrays.

The programming language APL [4] suggests two further operations on arrays that have a close relationship with the operations above. *Transposition*, ϕ is of type $\mathcal{A}^m(\mathcal{A}^n(\tau)) \rightarrow \mathcal{A}^n(\mathcal{A}^m(\tau))$ and satisfies the identity:

$$\{\{e \mid x \leftarrow e'\} \mid y \leftarrow e''\} \quad = \quad \phi\{\{e \mid y \leftarrow e''\} \mid x \leftarrow e'\} \qquad \text{provided } x \text{ does not occur in } e'' \text{ nor } y \text{ in } e' \tag{7}$$

The second operation, *reshape* $\rho(m, n)$, of type $\mathcal{A}^{mn}(\tau) \rightarrow \mathcal{A}^m(\mathcal{A}^n(\tau))$, builds a two-dimensional array. This is a right inverse to μ, i.e., $\mu(\rho(()m, n)(a)) = a$; we also have the identity:

$$\{e \mid x_i \leftarrow V\} \quad = \quad \mu\{\{e' \mid x_b \leftarrow W\} \mid W_a \leftarrow \rho(n, m)V\} \qquad \text{where } e' \text{ is obtained from } e \text{ by substituting } ma + b \text{ for } i \tag{8}$$

The m-shuffle operation needed in the fast Fourier transform and many other algorithms is simply defined as $\mu(\phi(\rho(m, n)V))$ for an mn-array V.

Lastly, we need to express the fact that we can interchange the order of summations over an $m \times n$-array. If W does not occur in e then

$$\Sigma\{\Sigma\{e \mid x_b \leftarrow W\} \mid W_a \leftarrow V\} \;=\; \Sigma\{\Sigma\{e' \mid x_a \leftarrow Y\} \mid Y_b \leftarrow \phi(V)\}$$
$$\text{where } e' \text{ is obtained from } e \text{ by interchanging } a \text{ and } b \quad (9)$$

Omitting a sign and a factor of $\sqrt{2\pi}$, *discrete Fourier transform* (DFT) of an mn-array V can be expressed as

$$\Phi^{mn}(V) \;=\; \{\Sigma\{\omega_{mn}^{ij}x \mid x_j \leftarrow V\} \mid i \leftarrow I^{(mn)}\} \quad (10)$$

In this, ω_n^i is the ith power of the nth principal root of 1, and $I^{(n)}$ is the array $0, 1, \ldots, n-1$. It is important to stress that (10) is a *program*, and one that can be written with minor syntactic changes in certain functional programming and database query languages. In such languages, the program would have a running time proportional to $(mn)^2$.

Using (8) and (9), we can transform the DFT (10) into

$$\{\Sigma\{\Sigma\{\omega_{mn}^{i(mb+a)}y \mid y_b \leftarrow z\} \mid z_a \leftarrow \phi(\rho(m,n)V)\} \mid i \leftarrow I^{(mn)}\}$$

Writing W for $\phi(\rho(m,n)V)$ and using (8) gives us

$$\mu\{\{\Sigma\{\Sigma\{\omega_{mn}^{(nd+c)(mb+a)}y \mid y_b \leftarrow z\} \mid z_a \leftarrow W\} \mid c \leftarrow I^{(n)}\} \mid d \leftarrow I^{(m)}\}$$

Moving a product over a sum in the obvious way gives us:

$$\begin{aligned}
& \mu\{\{\Sigma\{\omega_{mn}^{a(nd+c)}\Sigma\{\omega_n^{cb}y \mid y_b \leftarrow z\} \mid z_a \leftarrow W\} \mid c \leftarrow I^{(n)}\} \mid d \leftarrow I^{(m)}\} \\
= \; & \mu\{\vec{\Sigma}\{\{\omega_{mn}^{a(nd+c)}\Sigma\{\omega_n^{cb}y \mid y_b \leftarrow z\} \mid z_a \leftarrow W\} \mid c \leftarrow I^{(n)}\} \mid d \leftarrow I^{(m)}\} \quad \text{by (5)} \\
= \; & \mu\{\vec{\Sigma}(\phi\{\{\omega_{mn}^{a(nd+c)}\Sigma\{\omega_n^{cb}y \mid y_b \leftarrow z\} \mid c \leftarrow I^{(n)}\} \mid z_a \leftarrow W\}) \mid d \leftarrow I^{(m)}\} \quad \text{by (7)} \\
= \; & \mu\{\vec{\Sigma}(\phi\{\{\omega_{mn}^{a(nd+c)}t \mid t_c \leftarrow \{\Sigma\{\omega_n^{cb}y \mid y_b \leftarrow z\} \mid c \leftarrow I^{(n)}\}\} \mid z_a \leftarrow W\}) \mid d \leftarrow I^{(m)}\} \quad \text{by (5)} \\
= \; & \mu\{\vec{\Sigma}(\phi\{\{\omega_{mn}^{a(nd+c)}t \mid t_c \leftarrow \Phi^n(z)\} \mid z_a \leftarrow W\}) \mid d \leftarrow I^{(m)}\} \quad (11)
\end{aligned}$$

This last expression is a program that implements the DFT of size mn using m DFTs of size n. Having computed the latter, the number of additional operations used in (11) is readily seen to be proportional to m^2n. Thus if $T(n)$ is the time needed to compute the DFT of n, this transformation shows that $T(mn) = mT(n) + Km^2n$ for some constant K. When $m = 2$ we have the well-known recurrence relation $T(2n) = 2T(n) + K'n$, which shows that the number of operations needed to evaluate the discrete Fourier transform of an array of length n is proportional to $n \log_2 n$.

The foregoing does not complete the picture for arrays as collection types, nor does it tell us how to develop *really* fast Fourier transforms [3], however I believe it indicates the use of a common syntax for arrays and other collection types to be more than a notational convenience.

References

[1] R.S. Bird, "An Introduction to the Theory of Lists", *Logic of Programming and Calculi of Discrete Design*, M. Broy, ed, pp.3-42, Springer 1987.

[2] V. Breazu-Tannen, O.P. Buneman and L. Wong. "Naturally Embedded Query Languages", *Proc. Int. Conference on Database Theory*, Berlin, October 1992. Springer-Verlag LNCS, pages 140-154.

[3] O. Buneman, US Patent 4,878,187 “Apparatus and Method for Generating a Sequence for Sines and Cosines.”

[4] K. Iverson. *A Programming Language*, Wiley, 1962.

[5] G. Jones, “Deriving the fast Fourier algorithm by calculation”, *Glasgow Functional Programming Group Workshop*, K. Davis and J. Hughes, eds, Springer Workshops in Computing, 1990

[6] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201*, pages 1–16. Springer-Verlag, 1985.

[7] P. Wadler. “Comprehending Monads”, *Proc. ACM Conf. on Lisp and Functional Programming*, Nice, June 1990.

[8] D.A. Watt and P. Trinder. Towards a Theory of Bulk Types. Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, July 1991.