Programming with (finite) mappings

Peter Pepper, Bernhard Möller

Angaben zur Veröffentlichung / Publication details:

Pepper, Peter, and Bernhard Möller. 1991. “Programming with (finite) mappings.” In, edited by Manfred Broy, 381–405. Berlin: Springer. https://doi.org/10.1007/978-3-642-76677-0_28.

Programming with (Finite) Mappings

Peter Pepper Bernhard Möller

Abstract. We present a unified view of "mappings", abstracting from their appearance as "routines" (that is, objects that describe computations) or as "map-like data structures" (that is, objects that are stored and accessed via indices), respectively. On the basis of suitably defined algebraic operations we are then able to develop algorithms at a very high level of abstraction, without losing the potential of implementing them efficiently in terms of arrays and loops. This is demonstrated for a variety of array-based algorithms that are well-known from the literature.

1. Introduction

Traditionally, programming with array-like structures exhibits two characteristics: intricate index calculations and selective alteration. In this context, "selective alteration" means that our algorithms try – for the sake of space economy – to overwrite existing arrays rather than to generate new ones.

This view clearly originates from the imperative programming paradigm, and thus it is not surprising that array-like structures have hardly made their way into applicative programming – by contrast to many other structures such as sets, sequences, trees, and the like. The best that we can find in terms of abstraction are the assertional treatments in the style of Dijkstra (1976) or Gries (1980), sometimes enhanced by special operators (Jones 1980) or graphical symbolism (Reynolds 1981). First, tentative efforts to integrate array programs into the applicative style have been made only recently, for instance by Bird (1989) or Backhouse (1989).

Within the context of applicative programming, there is, admittedly, less need for array-like structures, because they may well be viewed as special cases of functions. Hence, in this area, programming with arrays is replaced by programming with higher-order functions.

Yet, this approach is satisfactory only from a theoretician's viewpoint. For, in practical applications, the distinction of special situations *is* a decisive feature. Even though we want to free the programmer from the burden of fiddling around

with the technical details of overwriting and the thus enforced intricate index calculations, we do want to keep the potential for efficient implementations. Thus, we encounter the need for array-like structures as a link between the too general concept of arbitrary higher-order functions and the too detailed concept of machine-oriented arrays. Technically, this link is established by a few fundamental transformation rules.

Moreover, even if we completely disregard the aspect of efficient implementation, there is still a need for increased abstraction in programming with functions. For, there are many "standard" situations where a single powerful operator (that is, a higher-order polymorphic function) can replace lengthy recursive function declarations. However, when designing a collection of relevant operators, we must be careful not to make this set too abundant, since, otherwise, there is the danger of losing comprehensibility.

Summing up, we aim at high-level algebraic operations that enable us to formulate specifications and algorithms as abstractly as possible. But we do this with a view towards implementation problems, in particular towards selective updating.

2. "Functions" and "Maps"

A "mapping" consists of three parts, the *domain*, the *range* (also called codomain), and the *graph*, that is, a set of pairs $\langle d, r \rangle$ with $d \in$ domain and $r \in$ range. As is well-known, the graph has to be "functional" (also called "left-unique"): no two pairs must share the same domain element d.

Given this basic definition, what distinguishes mappings in the mathematical sense from mappings in the sense of data structures? To ease the discussion about the differences, let us agree on the following phrasing: From now on, mappings in the mathematical sense will be called **functions**, whereas mappings as data structures will be called **maps**.

It is characteristic for a **function** that it is usually not defined extensionally by listing its graph, but rather *intensionally* by giving an algorithm that *computes* for any given argument x the corresponding result y. This algorithm usually is represented by some kind of (recursive) "function declaration", mostly written with the help of lambda abstraction, or as a set of conditional equations. Consequently, for a function f, the domain and range are usually determined only implicitly.

By contrast, a **map** is usually given *extensionally* by actually *listing* all pairs in the graph. As a consequence, the domain and range are specified explicitly as well. This extensional view entails the need for actually "generating" and "altering" maps, which are rare activities for functions.

In general, we do not distinguish the two concepts in the remainder of this paper, because the focus of our attention is the *method* of program development using maps. Hence, we describe the underlying theory only informally here. But we should point out that a rigorous semantic specification of the algebra of maps has to cope with a number of subtleties, in particular concerning questions of finiteness, definedness, strictness, and so forth. [1]

3. Notations

Our notations are essentially drawn from the set-theoretic view of mappings, but they are applicable under both the function view and the map view.[2] Let M, N be mappings, i, j be domain elements, and x, y be range elements. Then we use the following operations, which are all strict:

$\varnothing$	**empty map**.
$\downarrow M$	**domain** of M; (the "arguments" or "indices" of M).
$\#M$	**cardinality** of (domain of) M, that is, number of pairs in the map.
$M\ i$	**application** of M to argument i, yielding the corresponding value y (undefined if $i \notin \downarrow M$).
$M \mid D$	**restriction** of M to the set D; this is defined by $(M \mid D)\ i = M\ i$ if $i \in D$, $(M \mid D)\ i$ undefined otherwise. Note that we do not require D to be a subset of $\downarrow M$.
$M \leftarrow N$	**"overwriting"** of M by N; this is defined by the properties $(M \leftarrow N)\ i = N\ i$ if $i \in \downarrow N$, $(M \leftarrow N)\ i = M\ i$ if $i \notin \downarrow N$. Note that $\leftarrow$ is associative and idempotent.
$M \uplus N$	**"union"** of M and N; this is the same as overwriting but restricted to mappings which coincide on their intersection: $M \uplus N = M \leftarrow N$ if $M \mid (\downarrow M \cap \downarrow N) = N \mid (\downarrow M \cap \downarrow N)$, $M \uplus N$ undefined otherwise. Note that "$\uplus$" is associative, commutative, and idempotent.
$M \setminus\setminus N$	**constraining** of M by N; defined by $M \setminus\setminus N = M \mid (\downarrow M \setminus \downarrow N)$

[1] Formal definitions will be given in a forthcoming extended version of this paper.

[2] Since we merely want to give an informal sketch of the theory here, most of the subsequent equations only refer to the "standard" situations, where all map expressions are well-defined.

Using this operator, we can describe the overwriting operator concisely by
$M \leftarrow N = (M \setminus\setminus N) \uplus N$

$M \circ N$ **composition** of two maps; this is defined as usual by
$(M \circ N)\, i = M(N\, i)$.

The "**map comprehension**" also follows our set-oriented view[3]; that is, we essentially describe the association of a value to each index.

$[i \mapsto E[\![i]\!] \mid i \in D]$ mapping M with domain D, defined by $M\, i = E[\![i]\!]$ for $i \in D$, where $E[\![i]\!]$ is an expression in which i possibly occurs.

Sometimes we also enumerate small mappings by listing all pairs explicitly, in a form like $[i \mapsto E[\![i]\!], \ldots, j \mapsto E[\![j]\!]]$ or, following mathematical tradition, in the form of an indexed family $[x_i, \ldots, x_j]$.

This map comprehension differs from classical λ-abstraction essentially by its strictness: $E[\![i]\!]$ has to be defined for each $i \in D$; otherwise the whole map is undefined. In other words, we have the equation

$[i \mapsto E[\![i]\!] \mid i \in D] = \biguplus_{i \in D} [i \mapsto E[\![i]\!]]$.

The map comprehension is probably one of the most important concepts in our approach. For, it allows us to specify mappings in a pointwise fashion, without having to care about any computational ordering whatsoever.

Most approaches that strive for a more abstract and algebraic treatment of algorithms use a number of standard "homomorphism-oriented" operators[4]. In this spirit, we adopt the following notations:

$\hat{f}\, M$ "**lifting**"-operator[5]; the function f is applied to all elements of the range of M. (This "pointwise application" corresponds to Bird's "apply-to-all"-operator for sets and sequences.)

$\hat{f}\, M = [i \mapsto f(M\, i) \mid i \in \downarrow M]$

We extend this notation in a straightforward manner to situations where the function f results from fixing one argument of a binary operation. For example, if $fx \equiv a * x$, then $\hat{f}\, M$ is simply written as $a \hat{*} M$ (or just $a * M$, since we usually omit the lifting symbol).

---

[3] The notation is a straightforward extension of set-theoretic notations; it is also used by Jones (1980).

[4] This is very well elaborated by Bird (1987), but the theoretical foundation of this idea dates back at least to the work of von Henke (1975). Many of these concepts are also found in the language APL.

[5] We will omit the "^", that is, just write f instead of $\hat{f}$, whenever this kind of overloading can be resolved within the given context.

$\hat{f}\ (M, N)$ "**lifting**"-operator generalized to binary operations. The usual prerequisite here is, of course, that the two mappings share the same domain, but for the sake of flexibility we extend this operation also to the general case of non-fitting domains:

$$\hat{f}(M, N) = [i \mapsto f(M\ i, N\ i) \mid i \in (\downarrow M \cap \downarrow N)] \uplus M \setminus\setminus N \uplus N \setminus\setminus M\ .$$

f/M "f-**reduce**" over all values in the range of M, where f is a binary, commutative function. For instance, $+/M$ forms the sum of all values in M.

In many application areas – notably in linear algebra – the domains of the mappings are restricted to very special forms, namely to intervals $[a..b] \subseteq \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. We then often speak (by a slight abuse of language) of *vectors*. When the domain is a product of such intervals, we speak of *matrices*. (To ease readability, we adhere to the convention of denoting vectors by $\bar{a}, \bar{b}, \bar{c}, \ldots$ and matrices by $A, B, C, \ldots$) For these special domains, some further operators make sense:

$\bar{a} \phi \bar{b}$ "juxtaposition"[6] of vectors; the essential aspect is a proper shifting of the indices of $\bar{b}$ such that $\bar{b}$ is appended "right" of $\bar{a}$. Let $\downarrow\bar{a} = [a1..a2]$ and $\downarrow\bar{b} = [b1..b2]$; then:

$(\bar{a} \phi \bar{b})i = \bar{a}\ i$ if $i \in \downarrow\bar{a}$,
$(\bar{a} \phi \bar{b})i = \bar{b}(i - b1 + a2 + 1)$ otherwise.

Note that (although it is defined using indices) we mainly use this operator, when the concrete indices are of no relevance for the design of an algorithm.

$A \phi B$ "**horizontal juxtaposition**" of *matrices* A and B, that is, a renumbering of the columns of B.

$A \circ\!\!- B$ "**vertical juxtaposition**" of *matrices*; the rows of B need to be renumbered such that they directly follow those of A, that is, B is put "below" A.

In connection with matrices, that is, with mappings having direct products as domains, two further operations become a major issue, namely "Currying" and "transposing". We describe both operations for a given matrix $A : D_1 \times D_2 \to R$.

$\mathbf{C}A$ is the "Curried" mapping $A' : D_1 \to (D_2 \to R)$ defined by $(A'i)j = A(i, j)$.

---

6 We borrow the notation of Bird (1989).

$\bar{\mathbf{C}}A'$ is the "deCurried" map defined by
$\bar{\mathbf{C}}A'(i,j) = (A\ i)j$.

$\mathbf{T}A$ is the "transposed" matrix $B : D_2 \times D_1 \rightharpoonup R$, defined by
$B(j,i) = A(i,j)$.

Note: Sometimes we want to "Curry with respect to the second, rather than the first, component of the domain". This can easily be achieved by combining the two operators in the form $\mathbf{CT}A$, since we then have $\mathbf{CT}A\ j\ i = A(i,j)$. So we can easily express the two ways of viewing a matrix, viz. the row-oriented "ALGOL-view" and the column-oriented "FORTRAN-"view:

$Rows\ A \stackrel{\text{def}}{=} \mathbf{C}A$

$Cols\ A \stackrel{\text{def}}{=} \mathbf{CT}A$.

4. Implementation Issues

The operations and concepts presented in the previous chapter are oriented towards the design and development of high-level abstract algorithms. But we also want to keep an eye on efficient implementability. For, it is our claim that the elegance and clarity of applicative programs need not be in contrast with efficient execution. In the sequel we present a few transformation rules by which abstract map-oriented programs are converted into low-level loops. These rules are schematic enough to be used even in a fully automatic translation process.

The main task, here, consists in producing space-efficient code for allowing "in situ"-realizations of assignments $M := E[\![M]\!]$ for map variables M; these assignments arise in particular, when tail recursions are translated into loops. The following transformation rules[7] cover the majority of practically relevant cases. In the description of these rules, we restrict ourselves, for the sake of readability, to expressions E with at most two different applications of the mapping M. (For instance, the notation $E[\![M\ i]\!]$ shall indicate that no other applications of M besides $M\ i$ occur in the expression E.) The generalizations to more such applications as well as to the case of simultaneous updatings of several map variables are obvious.

The following rules shall give the principal ideas. Specific target languages may require specific additional constraints such as $D \subseteq \downarrow M$. (For instance, ALGOL68 arrays are more permissive than, say, Pascal arrays.)

[7] The correctness of these rules can be shown by Hoare's classical assertion logic or by the proof technique of Möller (1989).

Transformation T1: If the new value associated with each index i depends only on the old value $M\ i$, then the new mapping can be generated by successive (or parallel) overwriting:

$$M := M \leftarrow [i \mapsto E[\![M\ i]\!] \mid i \in D]$$
$$\Downarrow$$
for all i **in** D **do** $M[i] := E[\![M\ i]\!]$ **od**

(Note that the loop can be executed in arbitrary order, even in parallel.)

Transformation T2: If the new value associated with index i depends only on the old value $M\ i$ and some other value $M\ a$, then the new mapping can be generated by successive (or parallel) overwriting, provided that $M\ a$ remains invariant:

$$M := M \leftarrow [i \mapsto E[\![M\ i, M\ a]\!] \mid i \in D]$$
$$\Downarrow \quad \{E[\![M\ a, M\ a]\!] = M\ a$$
for all i **in** D **do** $M[i] := E[\![M\ i, M\ a]\!]$ **od**[8]

Transformation T3: If the new value associated with index i depends only on old values that belong to larger (smaller) indices, then the new mapping can be generated by ordered successive overwriting:

$$M := M \leftarrow [i \mapsto E[\![M\ i, M\ j]\!] \mid i \in [a..b]]$$
$$\Downarrow \quad \{a \leq b \wedge j \geq i$$
for all i **in** $[a..b]$ **do** $M[i] := E[\![M\ i, M\ j]\!]$ **od**

We have, thus, reached a state where we can stop our developments at a very high algebraic level, because the remainder of the code generation process can be left to an optimizing compiler. As a matter of fact, we may even stop as soon as the applicability conditions of the above transformations are met. Hence, our program derivations should be geared towards meeting the applicability conditions of these transformations.

5. Simple Exercises in Linear Algebra

A methodology for dealing with finite maps should, in particular, be able to cope with the classical matrix operations from linear algebra, such as scalar product,

[8] Alternatively, we could take a out of the domain of the loop; but this frequently makes the code even more costly, in particular in connection with parallel implementations.

matrix multiplication, Gaussian elimination, and so forth. So we start by reviewing some of these standard operations in our context.

5.1 Basic Operations

1. The simplest operator in linear algebra is the **scalar product** of two vectors, which we denote here by the infix symbol '$\cdot$'. Its mathematical definition is transcribed into our notation as pointwise multiplication (note that we omit the lifting symbol '$\hat{}$') and a subsequent +-reduce:

$$\bar{a} \cdot \bar{b} = +/(\bar{a} * \bar{b}) \quad (= \sum_i a_i * b_i) \tag{1.1}$$

Note that this operator is associative and commutative and, moreover, associates with the normal multiplication operator:

$$x * (\bar{a} \cdot \bar{b}) = (x * \bar{a}) \cdot \bar{b} \tag{1.2}$$

2. Slightly more complex is the **matrix product**[9], which we denote here by the infix operator '$\times$'. In linear algebra, it is usually described as follows: Let two matrices A and B be given such that $\downarrow(Cols\ A) = \downarrow(Rows\ B)$. Then the product is a new matrix each element of which is the scalar product of the i-th row of A and the j-th column of B. In our notation this reads

$$A \times B \stackrel{\text{def}}{=} [(i,j) \mapsto Rows\ A\ i \cdot Cols\ B\ j \mid i \in \downarrow(Rows\ A), j \in \downarrow(Cols\ B)] \tag{2.1}$$

3. Unfortunately, in many cases this definition is not very efficient as a program (namely within assignments of the kind $A := A \times B$), because none of the transformations from Section 4 is applicable. Fortunately, however, we can transform the definition of matrix multiplication further, such that transformation T1 becomes applicable. The trick lies in a suitable Currying:

$$\begin{aligned} Rows(A \times B) &= Rows[(i,j) \mapsto Rows\ A\ i \cdot Cols\ B\ j \mid i \in \downarrow(Rows\ A), \\ &\qquad\qquad j \in \downarrow(Cols\ B)] \\ &= [i \mapsto [j \mapsto Rows\ A\ i \cdot Cols\ B\ j \mid j \in \downarrow(Cols\ B)] \mid \\ &\qquad\qquad i \in \downarrow(Rows\ A)] \\ &= [i \mapsto Rows\ A\ i\ \hat{\cdot}\ Cols\ B \mid i \in \downarrow(Rows\ A)]\ . \end{aligned} \tag{3.1}$$

Here, every row of A is multiplied with all columns of B, yielding the corresponding row of the result matrix. To this version, transformation T1 is applicable, leading to the classical program for matrix multiplication.

---

[9] Boyle (1980) gives a derivation of these algorithms within the classical algebra of matrices.

5.2 LU-Partitioning by Gaussian Elimination

Next we should check, whether slighly more intricate algorithms can be treated as well. As a matter of fact, we show in this section that our approach renders also intricate numerical algorithms much more amenable to high-level programming concepts.

A standard algorithm in the area of numerical linear algebra is the solution of a system of linear equations by the method of Gauss.[10] Every mathematician or programmer working in this area is well familiar with the (quite ugly) FORTRAN programs that solve this problem. It is in particular the deep nesting of DO-loops in concert with the corresponding index calculations that makes these programs so hard to read, to verify, and to modify. So the question immediately comes to mind, whether this algorithm could not be *programmed* as nicely as it is *explained* in (good) textbooks. In order to keep the treatment brief, we demonstrate only a simplified version without pivot search.

1. The problem can be stated quite briefly: Given a matrix A, find a lower triangular matrix L and an upper triangular matrix U such that

$$A = L \times U \ .$$

2. The key to space efficiency here is that the constant parts of both matrices L and U can be omitted such that the relevant values can "share" one matrix layout. This is illustrated in the following diagram, where $\tilde{L}$ and $\tilde{U}$ denote the non-constant triangles of L and U, respectively:

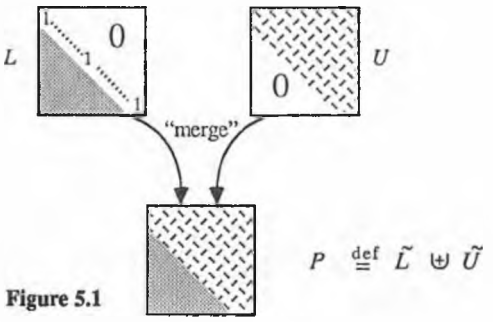


Figure 5.1

How can we realize this idea in our algebraic setting? A pair of mappings with disjoint domains is isomorphic to the union of these mappings, in the sense

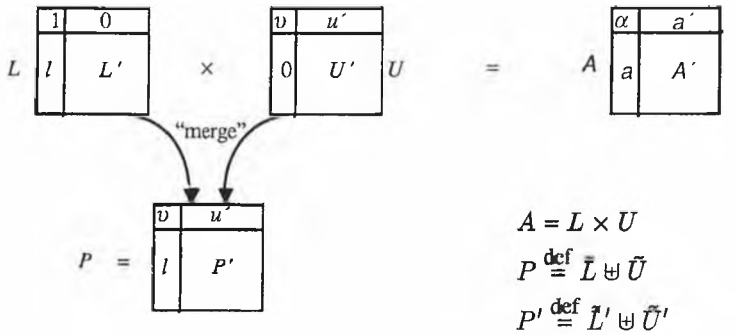
[10] We got the idea of using this example for the demonstration of our algebraic approach from a presentation given by Jim Boyle during the meeting of IFIP WG 2.1 in January 1989.

that we can reconstruct each of the original mappings, provided that we still know their domains. Formally[11]:

$$\downarrow A \not\cap \downarrow B \Rightarrow (A \uplus B) \mid \downarrow A = A,$$
$$\downarrow A \not\cap \downarrow B \Rightarrow (A \uplus B) \mid \downarrow B = B.$$

Hence, we can develop our algorithm for LU-partitioning based on the union of matrices rather than on their pairing.

3. The simplest version (from the point of view of understandability) of Gaussian elimination is illustrated by Figure 5.2 below, which reflects the following idea: We partition each matrix into four submatrices, namely the left upper element, the remaining first column, the remaining first row, and the remaining lower right matrix. Moreover, in L and U we ignore the constant parts that consist of 0's and 1's only.[12]



$$A = L \times U \tag{3.1}$$
$$P \stackrel{\text{def}}{=} \tilde{L} \uplus \tilde{U} \tag{3.2}$$
$$P' \stackrel{\text{def}}{=} \tilde{L}' \uplus \tilde{U}' \tag{3.3}$$

Figure 5.2

4. Our task is to find an algorithm *lup* that meets the specification

$$lup(A) = lup(L \times U) = \tilde{L} \uplus \tilde{U} \stackrel{\text{def}}{=} P \tag{4.1}$$

5. By standard matrix multiplication we can calculate for the above partitionings

$L \times U =$

v	u'
$l * v$	$l \times u' + L' \times U'$

$=$

α	a'
a	A'

$= A$

[11] By the symbol '$\not\cap$' we denote the disjointness predicate.

[12] We content ourselves with a graphical description of the partitionings, since they can be trivially defined using our restriction operator.

This entails the individual equations

$$\nu = \alpha \tag{5.1}$$

$$u' = a' \tag{5.2}$$

$$l = \frac{1}{\nu} * a = \frac{1}{\alpha} * a \tag{5.3}$$

$$L' \times U' = A' - l \times u' = A' - l \times a'. \tag{5.4}$$

For the triangular matrices L' and U' equation (4.1) yields again

$$\bar{L}' \uplus \bar{U}' = lup(L' \times U') = lup(A' - l \times a'). \tag{5.5}$$

6. The equations (5.1)–(5.5) determine the fragments from which the solution matrix $P = lup(A)$ is built up (for $A \neq \varnothing$):

$$lup(A) = a \uplus a' \uplus l \uplus lup(A' - l \times a') \tag{6.1}$$

$$\textbf{where } l = \tfrac{1}{\alpha} * a\ .$$

For the termination case $A = \varnothing$ we obtain, of course:

$$lup(\varnothing) = \varnothing\ . \tag{6.2}$$

7. Since "$\uplus$" is an associative (and commutative) operation with $\varnothing$ as neutral element, the standard *transformations for recursion removal* are applicable.[13] They result in the following version of our program:

$$lup(A) = lup_1(A, \varnothing) \tag{7.1}$$

$$lup_1(\varnothing, B) = B \tag{7.2}$$

$$\downarrow A \neq \varnothing \Rightarrow lup_1(A, B) = lup_1(A' - l \times a', B \uplus \alpha \uplus a' \uplus l) \tag{7.3}$$

$$\textbf{where } l = \tfrac{1}{\alpha} * a\ .$$

8. For this new function, we can immediately prove that the following *invariant* holds:

invariant $[\![lup_1(A, B)]\!] : \downarrow A \not\cap \downarrow B$.

That is, the domains of the two parameters are always disjoint. Hence, we can apply the "isomorphism" between pairing and union again and merge the two parameters into a single one while keeping the domain of A as a parameter in order to still enable the termination test.

Again, we have thus reached a stage, where an advanced optimizing compiler could take over and produce the efficient imperative code that is well-known from the FORTRAN programs in the pertinent libraries for linear algebra.

---

[13] An overview of such transformations as well as historical references can be found in the textbook by Bauer and Wössner 1982.

5.3 *Polynomial Interpolation*[14]

In textbooks on numerical analysis there is a relatively simple algorithm for polynomial interpolation, which has been designed by Aitken and Neville. The essence of this algorithm – from the point of view of numerical mathematics – is captured by a nice recursive equation. The tricky point is to program the evaluation of this equation in an efficient manner. For the sake of brevity, we merely sketch the idea of the algorithm here, concentrating on the essentials of the map-related developments rather than on the details of the numerical computations.

1. The essence of polynomial interpolation is given by a recursive function of the kind

$$P : nat \times nat \to real$$

which is specified by the equations

$$P(i, 0) = y\ i \qquad \text{for } i \in [0..N];$$
$$P(i, k) = e(P(i, k-1), P(i-1, k-1)) \quad \text{for } k \in [1..N], i \in [k..N]\ .$$

Here, y represents a collection of real values $y_0, \ldots, y_N$, and e is some complex numerical expression (the detailed nature of which is of no relevance for our considerations here).

The following diagram illustrates the basic principle of this computation:

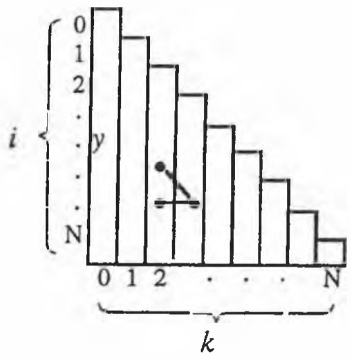


Figure 5.3

2. Since the second parameter function can be nicely inverted from $k-1$ to $k+1$, we can, again, apply a standard transformation rule for recursion removal. Hence, we prepare this transformation by Currying with respect to the second argument.

---

[14] Pepper (1981, 1984) already demonstrates a transformational development of this algorithm. Now, we want to show that more abstract algebraic operations can improve such developments.

This is achieved formally by introducing a new function

$$P_1 : nat \to (nat \to real)$$

with the definition

$$P_1 = \mathbf{CT} P = Cols\ P\ .$$

From this definition we can, then, deduce a directly recursive definition of P_1:

$$\begin{aligned} &P_1\ 0\ i = y\ i && \text{for } i \in [0..N]; \\ &P_1\ k\ i = e(P_1(k-1)(i), P_1(k-1)(i-1)) && \text{for } k \in [1..N], i \in [k..N]\ . \end{aligned}$$

Rewriting into map comprehension yields the form

$$\begin{aligned} &P_1\ 0 = y, \\ &P_1\ k = [i \mapsto e(P_1(k-1)(i), P_1(k-1)(i-1)) \mid i \in [k..N]]\ . \end{aligned}$$

3. Applying the aforementioned transformation rule leads to a tail-recursive version of our function, which has the functionality

$$P_2 : nat \times (nat \to real) \to (nat \to real)\ .$$

This function is defined by the equations:

$$\begin{aligned} &P_1(N) = P_2(0, y) \\ &P_2(k, p) = P_2(k+1, [i \mapsto e(p(i), p(i-1)) \mid i \in [k+1..N]]) \quad \text{for } 0 \le k < N \\ &P_2(N, p) = p\ . \end{aligned}$$

The map in the second equation allows the application of the transformation rule T3, because $i - 1 \le i$.

Again, we have reached a stage, where a compiler is able to produce an efficient imperative program that essentially consists of two nested loops.

6. Fast Fourier Transformation

Evaluating a polynomial

$$p(x) \stackrel{\text{def}}{=} a_0 + a_1 x + a_2 x^2 + \ldots + a_{N-1} x^{N-1}$$

of degree $N - 1$ at a specific point x_0 requires $O(N)$ operations. But many applications even need to evaluate p at N distinct points $x_0, \ldots, x_{N-1}$, which entails – if done naively – $O(N^2)$ operations. The algorithm of Cooley and Tukey (1965) reduces these costs to $O(N \log N)$ operations. It achieves this effect, however, only if the polynomial is to be evaluated at very specific points $r_0, \ldots, r_{N-1}$

(to which we will come in a moment). Thus, our task is to compute the values $y_0, \ldots, y_{N-1}$ defined by the equations

$$\begin{aligned} y_0 &\stackrel{\text{def}}{=} p(r_0) = a_0 + a_1 r_0 + a_2 r_0^2 + \ldots + a_{N-1} r_0^{N-1} \\ &\vdots \\ y_{N-1} &\stackrel{\text{def}}{=} p(r_{N-1}) = a_0 + a_1 r_{N-1} + a_2 r_{N-1}^2 + \ldots + a_{N-1} r_{N-1}^{N-1} \end{aligned} \tag{0.1}$$

for given vectors $\bar{r} = [r_0, \ldots, r_{N-1}]$ of arguments and $\bar{a} = [a_0, \ldots, a_{N-1}]$ of coefficients.

1. Algebraic properties from complex arithmetic. As usual, the actual programming has to be preceded by an analysis of the application domain, in our case trigonometric interpolation. The key concept, here, are the so-called "*N-th roots of unity*", which are complex numbers z having the property $z^N = 1$. It is known that for any N there are exactly N such numbers, one of which is called the "principal root of unity" and denoted here as *root*(N). Even though it is not necessary for our purposes, we note in passing that the principal root is obtained as $root(N) = e^{i2\pi/N} = \cos(\frac{2\pi}{N}) + i \sin(\frac{2\pi}{N})$, where i denotes the imaginary unit, that is, $i^2 = -1$. Then, all N roots are obtained by raising this principal root to the powers $0, \ldots, N-1$, that is, $r_j \stackrel{\text{def}}{=} r^j$. For later use, we record the following properties of the roots of unity:

Let $r \stackrel{\text{def}}{=} root(N)$; then:

$$r^0 = 1; \tag{1.1}$$

$$r^N = 1; \tag{1.2}$$

$$r^k \neq 1 \quad \text{for } 0 < k < N. \tag{1.3}$$

For even N, that is, $N = 2M$, we have moreover

$$r^M = -1; \tag{1.4}$$

$$r^{M+k} = -r^k; \tag{1.5}$$

$$r^2 = s \quad \text{where } s \stackrel{\text{def}}{=} root(M). \tag{1.6}$$

2. Now, we choose the N-th roots of unity as the evaluation points for our polynomial. That is, the argument vector in (0.1) is just

$$\bar{r} \stackrel{\text{def}}{=} [r^0, \ldots, r^{N-1}], \quad \text{where } N = \#\bar{a} \text{ and } r = root(N). \tag{2.1}$$

More generally, we define the vectors (for $0 \leq j < N$):

$$\bar{r}_j \stackrel{\text{def}}{=} [r_j^0, r_j^1, \ldots, r_j^{N-1}] = \bar{r}^j. \tag{2.2}$$

Now the original system of equations (0.1) can be written more compactly by using appropriate scalar products:

$$\begin{aligned} y_0 &= \bar{a} \cdot \bar{r}_0 \\ &\vdots \\ y_{N-1} &= \bar{a} \cdot \bar{r}_{N-1} \end{aligned} \tag{2.3}$$

3. If we introduce analogous vectors $\bar{s}$ and $\bar{s}_j$ for the M-th root of unity $s = root(M)$, where $N = 2M$ and $0 \leq j < M$, then properties (1.5) and (1.6) entail important relationships between the vectors $\bar{r}_j$ and $\bar{s}_j$. To this end, we need auxiliary operations even and odd that split a given vector into its even-indexed and odd-indexed components, respectively.[15]

$$even\ \bar{c} \stackrel{\text{def}}{=} [k \mapsto \bar{c}(2k) \mid 0 \leq k < \frac{\#\bar{c}}{2}]$$

$$odd\ \bar{c} \stackrel{\text{def}}{=} [k \mapsto \bar{c}(2k+1) \mid 0 \leq k < \frac{\#\bar{c}}{2}].$$

Now (1.5) and (1.6) immediately establish the relationships (for $0 \leq j < M$), where we omit – as usual – the lifting operator for the multiplication:

$$\begin{aligned} even\ \bar{r}_j &= \bar{s}_j & odd\ \bar{r}_j &= r_j * \bar{s}_j \\ even\ \bar{r}_{M+j} &= \bar{s}_j & odd\ \bar{r}_{M+j} &= -r_j * \bar{s}_j \end{aligned} \tag{3.1}$$

Note, how much we gain from properties (1.4)–(1.6), when N is an even number. Therefore, "in order to reap that benefit all through the computation, we shall restrict ourselves to the case that N is a power of 2" (Dijkstra 1984). Hence, from now on, we assume $N = 2^n$ for some n.

We derive the algorithm in two steps: First we deduce – based on the above properties – a **recursive solution** with the desired complexity of $O(N \log N)$ operations. Then, we implement this recursive algorithm **iteratively**, which entails an elaborate indexing of array elements in order to obtain a program that works **'in situ'**.

6.1 A Recursive Solution for the Fast Fourier Transform

4. The Fast Fourier Transform works for arbitrary polynomials p, and these polynomials are fully characterized by their coefficient vectors $\bar{a}$. On the other hand, the argument vector $\bar{r}$ derived from the N-th roots of unity is typical for the Fast Fourier Transform. Hence, we define the function *fft* that computes the result

[15] This corresponds to a rearrangement of the original polynomial into the form

$$p(x) = (a_0 + a_2 x^2 + \ldots + a_{N-2} x^{N-2}) + x * (a_1 + a_3 x^2 + \ldots + a_{N-1} x^{N-2})$$

vector $\bar{y} = [y_0, \ldots, y_{N-1}]$ of (0.1) and (2.3), respectively, for a given coefficient vector $\bar{a} = [a_0, \ldots, a_{N-1}]$, based on $r = root(N)$:

$$fft(\bar{a}) \stackrel{\text{def}}{=} [j \mapsto \bar{a} \cdot \bar{r}_j \mid 0 \leq j < N] \qquad \text{where } N = \#\bar{a} \,. \tag{4.1}$$

The essential gain of the Fast Fourier transform results from the application of a typical **divide-and-conquer** strategy. To realize this, we look for *common subexpressions* in the evaluation of (4.1), based, of course, on the above properties (3.1). To this end, let us consider the j-th equation; using the relationships in (3.1) we obtain for $0 \leq j < M = \frac{N}{2}$:

$$\begin{aligned} fft(\bar{a})j &= \bar{a} \cdot \bar{r}_j \\ &= (even\ \bar{a} \cdot even\ \bar{r}_j) + (odd\ \bar{a} \cdot odd\ \bar{r}_j) \\ &= (even\ \bar{a} \cdot \bar{s}_j) + r_j * (odd\ \bar{a} \cdot \bar{s}_j) \\ &= fft(even\ \bar{a})j + r_j * fft(odd\ \bar{a})j \,. \end{aligned} \tag{4.2}$$

An analogous derivation for the "other half" of points yields

$$fft(\bar{a})(M + j) = fft(even\ \bar{a})(M + j) - r_j * fft(odd\ \bar{a})(M + j) \,. \tag{4.3}$$

With the help of the juxtaposition operator we can combine these individual equations into the compact form

$$\begin{aligned} fft(\bar{a}) = (fft(even\ \bar{a}) + \bar{r}' * fft(odd\ \bar{a})) \,\phi\, (fft(even\ \bar{a}) - \bar{r}' * fft(odd\ \bar{a})) \\ \textbf{where} \quad \bar{r}' = [r_0, \ldots, r_{M-1}] \,. \end{aligned} \tag{4.4}$$

Hence, *fft* has the recursion structure

$$fft(\bar{a}) = h(fft(even\ \bar{a}), fft(odd\ \bar{a})) \,, \tag{4.5}$$

where the expression h is defined by

$$h(\bar{u}, \bar{v}) = (\bar{u} + \bar{r}' * \bar{v}) \,\phi\, (\bar{u} - \bar{r}' * \bar{v}) \,.$$

The common subexpressions $fft(even\ \bar{a})$ and $fft(odd\ \bar{a})$ entail the desired gain in complexity. Note also that the recursive calls of *fft* work on vectors that are half as long as the given argument vector.

Some further optimizations are possible, here. For instance, $\bar{r}'$ is based on the N-th root of unity $r = root(N)$, where $N = \#\bar{a}$. Since the computation of *root* is a costly process, we should use the relationship (1.6) in order to replace this operation by a cheaper one.[16] To this end, we simply add an additional parameter r to the function *fft* that invariantly has the value $r = root(\#\bar{a})$.

[16] This technique is known in optimizing compilers as "strength reduction"; in connection with program transformation it has been baptized "finite differencing".

5. For the termination case, that is, for $\#\bar{a} = 1$, we obtain from definition (0.1) the equation

$$\mathit{fft}([a_0]) = [a_0] \,. \tag{5.1}$$

Yet, for the sake of further optimization it is recommendable to stop the recursion one step earlier, that is, for $\#\bar{a} = 2$. For, here we need not compute the actual root r, since we merely use it in the form $r^0 = 1$.

$$\mathit{fft}([a_0, a_1]) = [y_0, y_1] \quad \textbf{where} \quad \begin{aligned} y_0 &= a_0 + a_1 \\ y_1 &= a_0 - a_1 \,. \end{aligned} \tag{5.2}$$

This concludes the derivation of the recursive solution of the Fast Fourier Transform, since the function has the envisaged complexity of $O(N \log N)$ operations.

6.2 An Efficient Iterative Implementation

The envisaged computational complexity can only be achieved if we "memoize" the values of the recursive calls instead of recomputing them repeatedly.

The above equations entail the following computational process: Initially, we need to calculate the values of one polynomial (at N points); in the next stage, we have to evaluate two polynomials (at $\frac{N}{2}$ points each); then four polynomials (at $\frac{N}{4}$ points each); then eight; ... and so forth. Consider, for instance, the computation tree of $\mathit{fft}(\bar{a})$ for the case $N = 8$, which is depicted in Figure 6.1 below. The left side indicates the even/odd rearrangement of the coefficient vectors, starting from the original vector $\bar{a}$, "on the way down into the recursion", whereas the right side shows the construction of the (intermediate) result vectors, using the operation h, "on the way back up from the recursion".

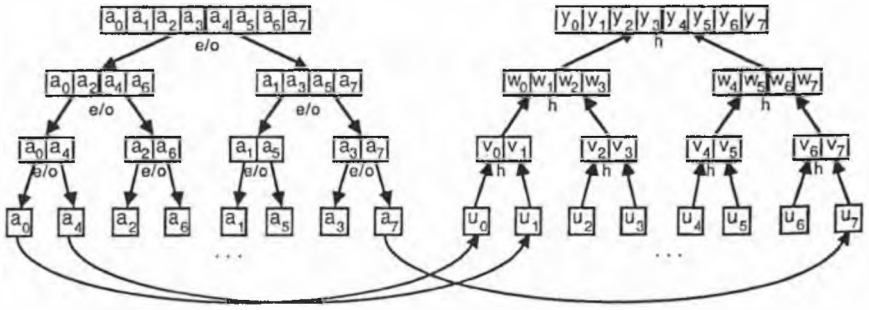


Figure 6.1

6. This is a tree-like recursion pattern that is amenable to general rules for recursion removal. (Since the pertinent rule is not found in the standard textbooks, we

present it in the Appendix.) The equations (4.4) and (5.1) are an instance of the scheme

$$f^{(k)}(x) = \Phi(f^{(k-1)}(\alpha\ x), f^{(k-1)}(\beta\ x)) \qquad \text{for } k > 0$$
$$f^{(0)}(x) = \Psi\ x\ .$$

if we interpret x as $\bar{a}$, $\alpha\ x$ as even $\bar{a}$, $\beta\ x$ as *odd* $\bar{a}$, Φ as h, k as $\log N$ for $N = \#\bar{a}$, $k-1$ as $\log(\frac{N}{2})$, and Ψ as id. Hence, the transformation from the Appendix yields the program (for $N = \#\bar{a}$):

$$\begin{aligned} \mathit{fft}(\bar{a}) &= C^k \circ S^k(A) \qquad (6.1)\\ &\textbf{where } A = [0 \mapsto \bar{a}]\\ &\qquad S = \mathit{Split}(\mathit{even}, \mathit{odd})\\ &\qquad C = \mathit{Comp}(h)\\ &\qquad k = \log(N) \end{aligned}$$

That is, we first perform $\log N$ even/odd splittings to the argument (as it is illustrated on the left-hand side of Figure 6.1) and then we perform equally many compositions using the expression h (as it is illustrated on the right-hand side of Figure 6.1)

Note, by the way, that the transformation rule from the Appendix converts the argument of the original function into a vector in the new function (in order to perform the "memoization"). Since the argument of *fft* is already a vector, we obtain through the transformation a vector of vectors, that is, a matrix. Moreover, the *even*/*odd*-splitting doubles the number of components of the given vector, but at the same time halves the size of each component vector, such that the overall size of the matrix remains invariant.

7. **Implementation of $S^k(A)$.** The next important observation now is that we can perform the first half of the computation ("down into the recursion") in one single step. It is easily seen that each element a_i ends up at a position p, the binary representation of which is just that of i read backwards. (For instance, in Figure 6.1 the element a_3 ends up at position 6, and 3_2 = '011' and 6_2 = '110'.)

The inductive proof of this fact depends on the following observation: Consider the map of maps A as a matrix, stored row-wise. Then the bitstring representation of the position of each element is of the form (where '⊣⊢' denotes concatenation of strings)

⟨⟨row number⟩⟩ ⊣⊢ ⟨⟨column number⟩⟩.

The odd/even splitting performed by the operation *Split* then moves the last bit of the column number to the end of the row number (thus doubling the number of

rows of the matrix and halving the number of columns). Ultimately, this process reverts each bitstring.[17]

Hence, $S^k(A)$ simply can be realized by swapping all pairs of elements $a_i \leftrightarrow a_j$ for which the binary representations of the indices i and j are the reverses of each other. This eliminates the need for recursion completely.

8. Implementation of C^k. Now we have to consider the "way back up from the recursion". But this simply means the iterated application of the operation *Comp* and, thus, of the operation h. Here, we observe from the definition (4.5) that h essentially maps a pair of two vectors (of size M each) into the union of two vectors (of size M each). This renders the generalized version of Transformation T1 applicable.

Hence, we have again reached a state, where a compiler could take over.

7. Transitive Closure of a Graph

Our method applies not only to linear algebra but to any area that makes use of mappings. A particularly nice example for demonstrating this effect is Warshall's algorithm for computing the transitive closure of a graph.[18]

1. We suppose that a directed graph over a fixed set N of nodes is given by a function

$$edge : node \times node \to bool .$$

This function is reflexive, that is, $edge(n, n) = true$ for all nodes n. Moreover, we use a predicate *path* that determines whether a given nonempty sequence of nodes is a path:

$$path\ p = \forall i \in [1..length(p) - 1] : edge(p(i - 1), p(i)) .$$

The following derivation also works, if we restrict ourselves to cycle-free paths. (Actually, some deductions become even simpler.)

2. Our task is to design a function

$$trans : node \times node \to bool$$

---

[17] Dijkstra (1984) uses these considerations about the matrix A as the framework for the derivation of the recurrence relations underlying the Fast Fourier Transform. But we feel that a clean separation of problem solving and implementation issues improves comprehensibility considerably.

[18] We follow here the principles of the earlier presentation by Broy and Pepper (1981), but now based on a considerably more abstract view.

that determines for any given pair of nodes, whether they are connected by a path in the given graph. So the specification of *trans* simply is (where "·" denotes concatenation of sequences and nodes)

$$trans(x, y) = \exists p : \textbf{\textit{sequ}}\ node \mid path(x \cdot p \cdot y) .$$

3. The underlying idea of the solution is to designate a set of "admissible paths". Initially, this set merely consists of the original edges, and then it is successively enriched until it contains all paths. The set of admissible paths in each stage is essentially controlled by the set of nodes that are allowed as "inner nodes" of the paths. This idea leads to the embedding into a new function

$$tc : set\ node \to node \times node \to bool .$$

This function is specified by

$$tc(S)(x, y) = \exists p : \textbf{\textit{sequ}}\ node \mid path(x \cdot p \cdot y) \wedge set(p) \subseteq S . \tag{3.0}$$

From this specification we can deduce the essential properties of *tc* (where '+' denotes addition of an element to a set; that is, $S + a = S \cup \{a\}$, with $a \notin S$):

$$tc(\varnothing)(x, y) = edge(x, y) \tag{3.1}$$

$$tc(S + a)(x, y) = tc(S)(x, y) \vee (tc(S)(x, a) \wedge tc(S)(a, y)) . \tag{3.2}$$

These two equations (which are easily provable from the specification, as is formally shown by Broy and Pepper, 1981) determine a terminating recursive algorithm. This algorithm solves the initially given problem, as is expressed by the following equation

$$tc(N)(x, y) = trans(x, y) \quad \text{where } N = \langle\langle \text{set of all nodes} \rangle\rangle . \tag{3.3}$$

Before we develop this algorithm further, we will, however, point out two immediate consequences of (3.2) that are helpful later on:

$$tc(S + a)(x, a) = tc(S)(x, a) \tag{3.4}$$

$$tc(S + a)(a, y) = tc(S)(a, y) . \tag{3.5}$$

4. At this stage, we want to emphasize the viewpoint that $tc(S)$ has a whole matrix as its result, and that this matrix is repeatedly overwritten during the evaluation process. Accordingly, we rewrite equations (3.1) and (3.2) into the equivalent forms (where we omit, for the sake of brevity, the domains of all maps, since they always are $N \times N$):

$$tc(\varnothing) = [x, y \mapsto edge(x, y)] \tag{4.1}$$

$$tc(S + a) = h_a(tcS) \quad \textbf{where} \tag{4.2}$$

$$h_a T \stackrel{\text{def}}{=} [x, y \mapsto T(x, y) \vee (T(x, a) \wedge T(a, y))]$$

5. From equations (3.4) and (3.5) we obtain the following properties for h_a:

$$h_a(T)(x, a) = T(x, a) \tag{5.1}$$

$$h_a(T)(a, y) = T(a, y) \tag{5.2}$$

This is precisely the enabling condition for transformation T2. Hence, the function h_a will lead to a loop, in which T is successively updated. And this loop is embedded into an encompassing loop that results from the function *tc* by application of some standard transformations for recursion removal (called "function inversion" by Bauer and Wössner 1982, because we start from the empty set $\varnothing$ and successively add elements until the full set N is reached).

As in all previous developments, this is a stage where we can leave the rest to the compiler.

Remark. The proofs in our derivation use only a few properties of the Boolean operations '$\wedge$' and '$\vee$', namely those of a *closed semi-ring* (cf. Aho et al., 1974). Hence, the development can be "parametrized" by the kind of underlying data type. For instance, it works analogously for the pair of operations '$\sqcap$' and '+' (where '$\sqcap$' yields the minimum of two numbers and '+' is addition), thus leading to an algorithm for *shortest paths*.

8. Conclusion

Programming with arrays, matrices, and the like has been regarded for a long time as that part of programming where the imperative style is, by necessity, superior to the applicative style. This view was justified as long as the attempts to program with such structures applicatively merely consisted in mimicking the imperative loops by tail-recursive functions. In that approach, the selective updating was only very clumsily represented by corresponding applicative operations.

However, as we have shown in this paper, a more radical transition to algebraic concepts immediately changes the situation in favor of the applicative style. By using a set of carefully chosen and powerful algebraic operators, we are able to program virtually all map-based programs in a much more compact and abstract manner than can be done in the imperative style. Above all, we get rid of the nitty-gritty index calculations that are so typical for this class of FORTRAN-like programs (which remain FORTRAN-like even when they are coded in Pascal or Ada).

However, the other extreme of "avoiding indices at all costs" (as it is done, for instance, in the approach of R. Bird, 1987, 1988) is, by our experience, not adequate either – at least in many applications. Hence, we have tried to find a proper balance between compact algebraic operators and explicit indexing.

A very important programming paradigm is "memoization", that is, the storing of calculated values instead of their recomputation. Our unified view of functions (as computations) and maps (as data structures) allows in particular a smooth transition between both concepts. Moreover, the high applicative level makes the derivation of parallel and sequential implementations equally natural and easy.

As a test for the usefulness of our approach, we have applied it in this paper to a wide variety of map-oriented programming problems. Most of these problems are standard in the literature[19], and some of them we had already treated by formal transformations in earlier papers. But, we feel that for all of them our present treatments are much better than the earlier ones – which demonstrates that high-level algebraic concepts constitute a decisive progress also in this area of programming.

Appendix: A Special Rule For Recursion Removal

Consider a tree-like recursion scheme of the kind

$$f^{(k)}(x) = \Phi(f^{(k-1)}(\alpha\ x), f^{(k-1)}(\beta\ x)) \qquad \text{for } k > 0,$$
$$f^{(0)}(x) = \Psi\ x\ ,$$

which leads to a balanced calling tree.

We can convert this tree-like recursion into a linear recursion by collecting all the nodes of each level into a vector. This leads to a computational process as illustrated below (for $k = 3$):

Here we have, for instance, $\bar{x}_1^{(2)} = \beta\alpha(x)$ and $\bar{y}_1^{(2)} = f(\bar{x}_1^{(2)}) = f(\beta\alpha(x))$.

On the "way down into the recursion" we produce the argument vectors, and on the "way back up from the recursion" we produce the intermediate result vectors. The former activity is performed by the operation

$$\mathit{Split}(\alpha, \beta)\bar{x} \stackrel{\text{def}}{=} [2k \mapsto \alpha(\bar{x}\ k), 2k+1 \mapsto \beta(\bar{x}\ k) \mid 0 \leq k < \#\bar{x}]\ ,$$

the latter activity by the operation

$$\mathit{Comp}(\Phi)\bar{y} \stackrel{\text{def}}{=} [k \mapsto \Phi(\bar{y}(2k), \bar{y}(2k+1)) \mid 0 \leq k < \tfrac{\#\bar{y}}{2}]\ .$$

Obviously, *Split* doubles the size of its input vector, whereas *Comp* halves it.

---

[19] Note that, in common textbooks, many examples that work on arrays are actually problems for sets or sequences. The arrays only creep in as a superficial implementation of these sets and sequences.

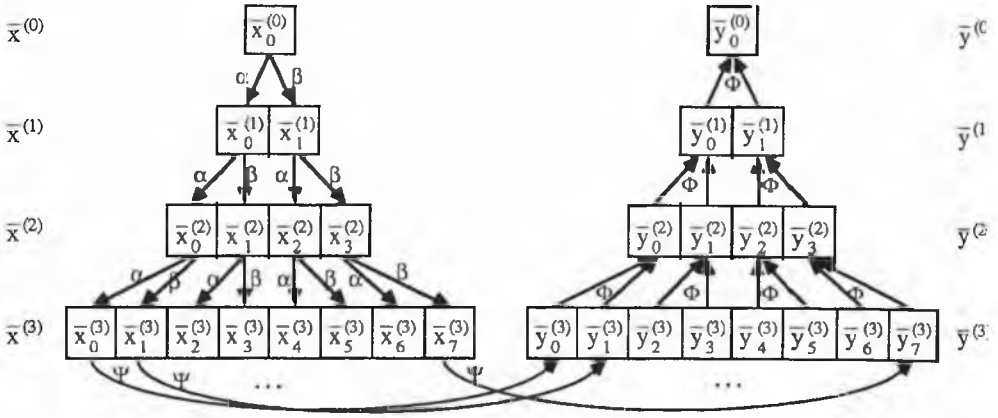


Figure A1

If we, now, introduce the definitions (for given k)

$$\bar{x}_{(0)} \stackrel{\text{def}}{=} [0 \mapsto x]$$

$$\bar{x}^{(i+1)} \stackrel{\text{def}}{=} \mathit{Split}(\alpha, \beta)\bar{x}^{(i)} \qquad \text{for } 0 \leq i < k$$

$$\bar{y}^{(k)} \stackrel{\text{def}}{=} \hat{\Psi}\bar{x}^{(k)}$$

$$\bar{y}^{(i)} \stackrel{\text{def}}{=} \mathit{Comp}(\Phi)\bar{y}^{(i+1)} \qquad \text{for } 0 \leq i < k$$

we can immediately show the following relationships

$$\bar{y}^{(i)} = \hat{f}\bar{x}^{(i)} .$$

This, in turn, means that

$$\begin{aligned} \bar{y}^{(0)} &= \hat{f}\bar{x}^{(0)} \\ &= (C^k \circ \hat{\Psi} \circ S^k)\bar{x}^{(0)} \\ &\quad \textbf{where } S = \mathit{Split}(\alpha, \beta) \\ &\qquad\quad C = \mathit{Comp}(\Phi) . \end{aligned}$$

Since each vector $\bar{x}^{(i)}$ and $\bar{y}^{(i)}$ has exactly 2^i elements, we have in particular that

$$\bar{y}^{(0)} = [0 \mapsto y] \text{ and } \bar{x}^{(0)} = [0 \mapsto x] \quad \text{with } y = f^{(k)}(x) .$$

In other words, we start by embedding the argument x into the singleton vector $[0 \mapsto x]$, apply k times the operation $\mathit{Split}(\alpha, \beta)$, once the (pointwise lifting of) the operation Ψ, and again k times the operation $\mathit{Comp}(\Phi)$. The result is a singleton vector consisting of the desired result $y = f^{(k)}(x)$.

Acknowledgement. The ideas presented here were strongly influenced by many discussions with members of IFIP WG 2.1, notably with R. Bird, L. Meertens, and J. Boyle. We are grateful to U. Berger and F. Erhard for helpful comments. We also enjoyed clarifying disputes with M. Broy on variations of the algebra of maps.

References

Aho, A.V, Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Reading: Addison-Wesley 1974.

Backhouse, R.: Making Formality Work For Us. Bulletin of the EATCS 38, June 1989, 219–247.

Bauer, F.L., Wössner, H.: Algorithmic Language and Program Development. Berlin: Springer 1982.

Bird, R.S: An Introduction to the Theory of Lists. In: Broy, M. (ed.): Logic of Programming and Calculi of Discrete Design. Proc. Int. Summer School, Marktoberdorf 1986, NATO ASI Series F, Vol. 36, Berlin: Springer 1987, 5–42.

Bird, R.S.: Costructive Functional Programming. In: Working Material, Int. Summer School on Constructive Methods in Computing Science, Marktoberdorf 1988.

Bird, R.S., Wadler, P.: An Introduction to Functional Programming. Englewood Cliffs: Prentice-Hall 1987.

Boyle, J.: Towards Automatic Synthesis of Linear Algebra Programs. In: Hennell, M.A., Delves, L.M. (eds.): Production and Assessment of Numerical Software. London: Academic Press, 1980, 223–245.

Broy, M., Pepper, P.: Program Development as a Formal Activity. IEEE Transactions on Software Engineering SE-7, 1 (1981), 14–22.

Burstall, R.M., Darlington, J.: A Transformation System for Developing Recursive Programs. J.ACM 24 (1977), 44–67.

Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Math. Comput. 19 (1965), 297–301.

Dijkstra, E.W.: A Discipline of Programming. Englewood Cliffs: Prentice-Hall 1976.

Dijkstra, E.W.: The Fast Fourier Transform and the Perfect Shuffle. In: Working Material, Int. Summer School on Control Flow and Data Flow, Marktoberdorf, 1984.

Gries, D: The Science of Programming. New York: Springer 1980.

von Henke, F.W.: On Generating Programs from Types: An Approach to Automatic Programming. In: Huet, G., Kahn, G. (eds.): Construction, Amélioration et Vérification des Programmes. Colloques IRIA 1975, 57–69.

Jones, C.B.: Software Development: A Rigorous Approach. Englewood Cliffs: Prentice-Hall 1980.

Möller, B.: Applicative Assertions. Proc. Int. Conf. on Mathematics of Program Construction, Groningen 1989.

Pepper, P.: Specification Languages and Program Transformation. In: Reid, J.K.(ed.): Relationship between Numerical Computation and Programming Languages. Proc. IFIP WG 2.5 Conf., Boulder 1981. Amsterdam: North-Holland 1982, 331–346.

Pepper, P.: Inferential Techniques for Program Development. In: Pepper, P. (ed.): Program Transformation and Programming Environments. NATO ASI Series F, Vol. 8. Berlin: Springer 1984, 275–290.

Pepper, P. (ed.): The Programming Language Opal. Internal report, Technische Universität Berlin 1991.

Reynolds, J.C.: The Craft of Programming. Englewood Cliffs: Prentice-Hall 1981.
Smith, D.R.: KIDS – A Knowledge-Based Software Development System. In: Proc. of the Workshop on Automating Software Design, AAAI 1988.