OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

The Applicative Style of Programming

David S. Wise

Computer Science Department
Indiana University
Bloomington, IN 47405

Computer Science Department
Oregon State University
Corvallis, OR 97331

1984-2

The Applicative Style of Programming

by David S. Wise

1. Introduction

Pretend for the moment that you know nothing about programming and just consider algorithms. Algorithms, sets of clear instructions on how to produce given results, outputs, or behaviors, are central to computer science. That is, the science of computing pivots around algorithms. Before one even discusses programming---or styles thereof---one must confront the underlying problem of just what is it that you want a machine to do.

In the next paragraph I shall abandon discussions of various algorithms for a while to consider style, but note from the onset that there exist both good and bad algorithms for solving any (solvable) problem, and that an individual algorithm may be expressed in all decent programming languages, albeit with varying degrees of clarity. (Furthermore, different machine architectures expand our perception of "good" and "bad.") Most certainly algorithms are not equivalent to the individual computer programs (in individual languages) that might implement them.

This article explores a less conventional way of expressing algorithms, called applicative or functional programming. Both of these monikers are punny, for what kind of programming is interesting if it does not apply to some problem or does not work, or function, in an effective manner? The intention of the dual title is neither of these droll misinterpretations.

I mean, by applicative or functional programming, a pure style of expressing algorithms as mathematical functions that map from arguments (usually called input) to results (output). The only control structure is application of defined functions to specified arguments. The only output from a functional program is the results of some such "outermost" function. The only binding---or interpretation of "variables"---arises from association of argument with abstract parameter. Interestingly, and we shall see this to be very important later, there need not be much explicit specification of sequentiality or order-of-action over time, except that implicit from mathematical depending of results (indirectly) upon arguments.

2. Arithmetic

A fine example of functional programming is familiar to all: standard infix notation of arithmetic expressions. Examples are addition,

subtraction, and multiplication:

(1 + 3) ; (4 - 7) ; (3 x 5) ;

or, to use a definition from the formal study of language:

E ::= (E + E) | (E - E) | (E x E) | integer .

This rule is read that an E (expression) is defined to be either an addition, subtraction, multiplication, or a simple integer, where the terms/factors in the more complicated expressions are also (recursively) E's; a few alternative definitions of E are given below. Like well-formed-formulas in logic, however, there will only be a precise number of alternatives in any language.

The semantics of this language is well-refined and perhaps too conventional, because we often fail to see the general ideas in such a specific and familiar case. Two features of it are that the function (or operator) takes exactly two arguments (i.e. that it is binary, or dyadic) and that it is conventionally written between the arguments (or operands) in infix notation. Their third shared feature, which has been relaxed in only a few higher-level programming languages, is that the result of a function is a single, atomic value, here an integer. The term, unate, will specify this property of an operator or function.

The necessity for a function being unate is easiest to shatter; consider integer division. Conventionally (and I mean primary school arithmetic) integer division has two results: quotient and remainder. So shall it be here; integer division is to be both binary and binate, returning two results. This convention, however, requires us to agree on some way of representing non-atomic values; elsewhere one uses arrays, vectors, records, register-files, etc. (Let us avoid a unnecessary tangent on data-structures at this point.) I choose what I believe to be the simplest, a list structure, and define it below. For now, let us agree that a two element list may be constructed using angle brackets around two arithmetic expressions:

⟨8 4⟩ ; ⟨(1 + 7) (2 x 2)⟩ ,

and decomposed into either of its two elements by applying the appropriate probing (projection) function first or second to it. Then we would unambiguously introduce integer division by requiring that its result be a list of two integers: quotient and remainder, respectively. Thus,

E ::= (E / E) ;

and

(35 / 3) = ⟨11 2⟩ ,

and what we see in most languages for integer division becomes a composition of first and (/)

3. Prefix notation

My, but that was hard to write! Just what was the last thing in the previous section? Well, you just saw me trip over infix notation. (I mentioned earlier that it is not sufficiently general.) Avoiding an

analysis of why infix becomes awkward, allow me to repair the problem by introducing left-hand (or prefix) functional notation. Define expressions, E as before, to be either integer or list valued. Elist can be a string of E's, and F is a function.

```
E     ::=   F : E  |   ⟨ Elist ⟩ | integer  ;
Elist ::=   empty  |   E Elist  ;
F     ::=   plus   | minus       | times     | divide .
```

If this new syntax replaced (rather than just extended) the previous one, the only syntactic symbols that would remain are the angle-bracket list builder and the infix colon, indicating function application. The examples above would then translate to

```
plus:⟨1  3⟩ ;          minus:⟨4 7⟩ ;          times:⟨3 5⟩ ;
divide:⟨35 3⟩          =          ⟨11 2⟩      .
```

Actually, we can safely retain the infix notation as a syntactic convenience for certain (very familiar) operators as long as they have alternate prefix names like these.

Now I can reiterate the last sentence from the previous section. FORTRAN, PASCAL, et fils abandon the primary school definition of integer division by neglecting the remainder (likely available in hardware anyway). That is, in those languages division is what we expect from "composing" first and divide, as defined here. The problem before was that we had no crisp (prefix) name for the division operator. Exclusively represented by a semi-syntactic symbol that necessarily sat in infix position and between required, wrapping parentheses, the division function had no clear name for use here in running English. Among all that other syntax, it wasn't quite right to call it merely "/". (Denying certain functions a general syntax in favor of an embellished syntax, often raises such awkwardnesses upon subsequent generalization.)

I shall stick with spelled-out names (like identifiers') for unfamiliar functions in this paper. This convention has the desirable pragmatic effect of requiring that new functions be given names as we define them. If we choose the names to be meaningful, as well as pronounceable, we practice good engineering, documenting our creations as we create. While it is good to put only a formal name on an unfamiliar beast, familiar animals may enjoy the additional luxury of a syntactic nickname.

4. Conditional Expressions

We need an important feature of ALGOL 60 expressions that was lost in successors to that language, the conditional expression. It is introduced here as the primitive if that takes an odd number of arguments. When applied to a singleton argument, if behaves like the identity function:

```
if:⟨elsepart⟩  =  elsepart  .
```

Most familiar is the form when if is applied to a triple:

```
if:⟨truthvalue thenpart elsepart⟩  .
```

This is the conventional conditional expression from ALGOL 60 which asks

that truthvalue be evaluated either to true or to false. In the first instance it evaluates to whatever thenpart evaluates to; in the other case it evaluates to the value of elsepart.

Extensions to quintuples, septuples, etc. extend this pattern. Predicates (or truth-valued expressions) occur in odd-numbered positions, except the last; possible-result expressions occur in the immediately following even-numbered positions. The truth-values are evaluated in order until the first true one is found; the value of the conditional is the value of the expression immediately following. If all truth-values. are false then the last expression in the argument list determines the value of the conditional expression.

In order to make decent use of conditional expressions, we shall need some non-trivial predicates, or truth-valued functions. I introduce three familiar ones without much explanation, except to notice that they, too, are familiar binary, unate, infix operators, and therefore are allowed a their common infix notations.

```
F  ::=    less? |    greater? |  equal? ;
E  ::=    (E < E) |  (E > E) |  (E = E) |  TRUE |      FALSE .
```

It is possible to define these "order" predicates over any well-ordered data type; here, we intend that integers be compared to integers.

In introducing these two values for truth, either we may require them to be of a distinct boolean type, or we might interpret their values as integers: say, false as 0 and true as 1 or another positive integer. The choice does not matter here, so picture a truth-value in whichever way you like. Like internal representation of numbers, this decision is best left under-specificed so that an implementation of the language is less constrained, better to fit a peculiar machine.

5. Recursion for Creation

Now the tools are in hand for a little creative programming. The rules of the game are, from Section 1, that the only control is application of functions to arguments. How can functions be applied with unforeseen nesting? The answer, all too fearsome to traditionally-trained programmers, is recursion. All readers, however, have forgotten how to program for the moment, so none are intimidated by this yet unknown term. Let us discover it.

The essence of recursion is the idea that a function exists before we set about writing a program for it. All we need to do is to specify which program it is that we desire. Indeed, any program for a particular function---say, greatest-common-divisor---is merely one of many possible descriptions for the function, which "becomes" the function with the aid of some hardware.

5.1. A Problem

First, we need a problem that we can treat with the few primitive functions in hand. Arithmetic, integers, comparisons, conditionals, hmmmmmmmm; why not "greatest common divisor" or gcd? (I'll bet you picked that one too!)

The greatest common divisor of two integers is the largest integer that evenly divides them both. For now we shall consider both positive and negative integers although the result of gcd must surely be positive. (Why?)

5.2. Some Algebra

Now let's tinker a bit with mathematics; here are some identities that always apply.

```
gcd:⟨i j⟩ = gcd:⟨i -j⟩   since sign does not affect definition;        (1)
gcd:⟨i j⟩ = gcd:⟨j i⟩    from commutativity of gcd definition;         (2)
gcd:⟨i j⟩ = gcd:⟨(i+j) j⟩          discussed below.
```

Equation (3) holds because if

```
g = gcd:⟨i,j⟩
```

then g evenly divides both i, j, and therefore their sum. Moreover, if any integer n, $n \geq g$, divided both (i+j) and j, then n would necessarily be a divisor of i, and therefore $n \leq$ gcd:⟨i,j⟩ =g; hence n=g . From these three equations many others may be derived. Of them, only three turn out to be of interest below.

```
gcd:⟨i j⟩ = gcd:⟨(j-i) i⟩                    from (2), (1), (3), and (1);
gcd:⟨i j⟩ = gcd:⟨(i-j) j⟩                    from (1), (3), and again (1);
gcd:⟨i j⟩ = gcd:⟨second:divide:⟨i j⟩ j⟩      by (2) or (5) repeatedly.[1]
```

Justifying Equation 6 requires a case analysis, which we'll skip because it turns out to be uninteresting.

5.3. Simple Cases

The form of a good recursive definition usually takes the form of

```
gcd:⟨i j⟩ =: if:⟨      .......      ⟩ .
```

where the equals-colon symbol is used in the infix position to suggest a definition, and the ellipsis in the conditional expression is to be filled in. (Section 9 offers the exception to this rule.) Assuming that integer equality is available, we might start out

```
gcd:⟨i j⟩ =: if:⟨      (i=j)  i
                       .......      ⟩ .
```

This would be fine when i is positive, but we must not neglect the possibility that i and j may be negative---or even zero. How about using Equations 1 and 2 to eliminate these exceptional cases first?

---

[1] From the rules of Section 3 we infer that colon "associates to the right" like the conventional exponential operation. Thus:

```
divide:divide:⟨139 8⟩   = ⟨5 2⟩ .
```

```
gcd:⟨i j⟩ =: if:⟨      (i⟨0)  gcd:⟨(0-i) j⟩
                       (j⟨0)  gcd:⟨i (0-j)⟩
                       (i=0)  j
                       (j=0)  i
                       (i=j)  i
                       .......        ⟩ .
```

That takes care of the touchy cases; at this ellipsis we know that $0 \leq i \neq j > 0$ and that we have defined the correct result in other cases. Now we can use Equations 4 and 5 from above.

5.4. Other Cases

In order to direct our thinking toward a non-obvious (but useful) algorithm, let us assume that we are to implement gcd on a machine that does not have efficient division or multiplication. This is a surprisingly frequent constraint; hardwired multiplication can be painfully expensive and hardware division is often not be available at all. Equation 6, which suggests that gcd be recast using remainder on integer division, is therefore skipped.[2]

Let us try to complete the definition of gcd. with Equations 4 and 5, applied in a way to keep differences positive:

```
gcd:⟨i j⟩ =: if:⟨      (i⟨0)  gcd:⟨(0-i) j⟩
                       (j⟨0)  gcd:⟨i (0-j)⟩
                       (i=0)  j
                       (j=0)  i
                       (i=j)  i
                       (i⟨j)  gcd:⟨(j-i) i⟩
                       (i⟩j)  gcd:⟨(i-j) j⟩
                       .......        ⟩ .
```

A bit of logic tells us that the ellipsis is now irrelevant because of the total ordering on integers.

Congratulations! We have just reinvented the simple form of Euclid's Algorithm.

5.5. Is it Correct?

The formulation now has the property that we have been searching for. It is total or well-defined on all integer input. This takes a bit of proof, which might appear to an experienced programmer as some sort of testing. The similarity is no accident, because recursive programming lends itself to proof by a kind mathematical induction, known as recursion-induction, that is very nearly the same mental exercise as creating the program in the first place.[3] If the reader is squeamish

---

[2] It might be useful on hardware where integer division is efficient, or on data where division by repeated subtraction is uniformly less efficient than other division algorithms; we ignore these possibilities for now.

[3] While writing programs that can be supported by correctness proofs is desirable, pro forma proofs are not necessary in programming prac-

about proofs, this section is easily skipped on first reading.

Recursion-induction, a term coined by John McCarthy, works (roughly) like this. First, establish that this code for gcd is defined for i=0=j, and then for i=0 or j=0; we did this earlier. Then, hypothesize that this gcd is well defined for i and j positive and less than some number, n; prove that it is also well-defined for i and j positive and less than n+1 by using the observation that either

$$i = j = n+1 = gcd:\langle i\ j\rangle$$

or, after one reduction step using one of the alternatives introduced last (from Equations 4 and 5), the hypothesis applies. Finally, observe that if either i or j (or both) are negative, one or two reductions expresses the expression in terms of gcd with non-negative arguments---which we have just considered.

The previous paragraph, with the proof of Equations 1-5, above, constitutes a proof of strong correctness of this definition of gcd. To the mathematician, this means that the expression constructively specifies the function as defined originally. To the computer scientist, this means that the expression may be interpreted as a program mechanically and, for any integers i and j, it will stop and return the correct answer.

In fact, the two interpretations coincide, and that's a good thing because the underlying philosophical concepts of correctness are the same. That the two concepts of proof unify with a program expressed under this applicative style of programming is quite a powerful observation, because it means that a programmer can construct a program and prove it at the same time. If he bothers to write his programs in such a language with such rigor, then he produces a much refined product. It will be correct (in the first place), easy for others to read and follow, and thereby easier for them to maintain or revise, should specifications change.

I do not claim that these virtues occur automatically simply by adopting this kind of language; I do claim that, because it is closer to the style of mathematics (as refined over the centuries), this language facilitates this kind of disciplined programming. It still takes a disciplined programmer to follow through on rigorous thought patterns; I claim that this style does enable more people to practice that discipline.

Another way to decide whether one's program is wrong is to "run" a few cases. No further rules are necessary about how this program actually works, but I do like to verify how the reduction works, if only for myself:

---

tice. For a provocative review of the role of proofs in programming (and in mathematics), see the essay by De Millo, Lipton, and Perlis on Pages 271-280 of the 1979 Communications of the ACM (Vol. 22), with responses from readers on Pages 621-630 of that same volume.

```
gcd:<78 -21>  =  gcd:<78 21>  =  gcd:<57 21>  =  gcd:<36 21>
              =  gcd:<15 21>  =  gcd:< 6 15>  =  gcd:< 9  6>
              =  gcd:< 3  6>  =  gcd:< 3  3>  =              3 .

gcd:< 0 -21>  =                  gcd:< 0 21>  =             21 .

gcd:<78   0>  =                                             78 .
```

Not only might this uncover cases where a definition is incorrect, but also it exercises the cases that arise in an inductive proof. While such testing is not a proof, it generally helps to identify the cases that must be considered in one.

Even when a proof is not explicitly included as documentation, thought patterns, like this, that generate an applicative program constitute a weak proof of its correctness. A good programmer unconsciously follows the course of a recursion-induction proof as he programs, and that proof is almost visible in his code.

5.6. Refinement for Clarity/Efficiency Now that we have a correct program, let us rearrange it syntactically. The first change might have happened in constructing gcd two sections back. At that point we might have split the problem into two functions: one for integer arguments and one for positive-integer arguments. The result looks like this:

```
gcd:<i j> =: if:<      (i<0)  gcd:<(0-i) j>
                       (j<0)  gcd:<i (0-j)>
                       (i=0)  j
                       (j=0)  i
                       GCD:<i j>  >   .

GCD:<i j> =: if:<      (i=j)  i
                       (i<j)  GCD:<(j-i) i>
                       (i>j)  GCD:<(i-j) j>
                       .......          >  .
```

We needed another name for the second function above, and I used uppercase so that the proof in the previous section still scans. In this case the GCD function is conceptually internal to the definition of gcd, and not to be called directly from outside gcd.

Not only is this a good place to split our thoughts, it is a good place to split the computation, because GCD need not call gcd. Therefore, most reduction is accomplished through GCD which, with constrained arguments, has become simpler (and faster).

As observed earlier, the ellipsis in GCD may be omitted; in fact, the order of its predicate-value lines may be scrambled because they are mutually exclusive. We are free to choose any of the six permutations and to omit the (respectively) last test; the last result becomes an "elsepart" of the conditional.

```
GCD:<i j> =: if:<      (i<j)  GCD:<(j-i) i>
                       (i>j)  GCD:<(i-j) j>
                       i                  >  .
```

I chose this form for elegance and for efficiency. The two predicates

that remain are nicely symmetrical in appearance, and this order of testing postpones the least likely alternative to the most remote position, the "elsepart". It does, however, move the "simple" case to a position deeper in the program. (The reader should appreciate that this program would never be initially composed in this cleansed.)

Further efficiencies might be necessary, depending on an implementation. One might avoid writing two subtractions like this:

```
GCD:<i j> =: if:<      (i<j)      GCD:<j i>
                       (i>j)      GCD:<(i-j) j>
                                  i                    > .
```

This is really an algebraic transformation using Equation 2. Other transformations are possible, including some that transform GCD into BASIC-like code for use under traditional iterative, languages.

5.7. Summary of Creating Recursions This section has become terribly long, and only treated a simple example. Someone who has forgotten how to program, however, probably needs a review of what just happened. We learned how to program recursively in just five steps.

First, we formulated an intuitive understanding of the desired function and we assumed that it did exist. Then (second) we set out to describe that extant function by specifying its behavior on simple input arguments; for each elementary case we described the complete answer. The trick is, of course, to be able to isolate those simple case using simple predicates.

Having handled the simple cases, thirdly, we looked for reductions of each of the complicated cases to other invocations of the available functions, including the one under construction. The idea here is that we can safely apply the new function (here gcd) to a case slightly less complicated than the one being confronted. In the case of gcd, we reduced the complicated case to that of taking gcd of slightly smaller integers. Then with a prototype definition in hand, we tested it or---better yet---proved it as the fourth step.

Finally, the code was polished with consideration for readability and efficiency. By using algebraic transformations here, we are sure to preserve the correct function as we alter the form of its specification. These last few steps do not necessarily proceed so smoothly, because late insights might prompt revisions to early work, with succeeding steps necessarily repeated.

For instance, testing may uncover a case not properly handled or polishing might reveal a generalization of the function that would make it useful for cases other than that which immediately motivated the programming effort. It is also possible that predicates for reductions of complex cases may not be readily available. In such cases, additional "helper" functions, like GCD, might be required to complete the original program.

These are familiar problems, however. Most problems are not refined into tractable subproblems on first analysis, and most programs get redesigned after some testing. These are ordinary steps in "stepwise refinement," a powerful tool that fits nicely into applicative

programming.

All you need to do is to assume that the program (function) exists, and then set it down on paper. The only difference is that under applicative programming, all the hard problems may be postponed to the end---instead of to the middle of the program under construction.

6. List Processing

Section 2 introduces the concept of a list, restricted to a list of two values: a pair. This section generalizes that concept to a list of any length, including (surprisingly) infinity.

Let us first define the list handling primitives by extending F.

```
F ::=   first   | rest   | null?   | atom?   | cons .
```

The first is easiest to explain because we have already seen it; first extracts the leftmost element from a list:

```
first:⟨8 4⟩ = 8 .
```

Rest returns the remainder of the list with the first element removed:

```
rest:⟨8 4⟩  = ⟨4⟩ ;
rest:rest:⟨8 4⟩      = ⟨⟩ ;
first:rest:⟨8 4⟩     = 4 .
```

From the last example, we see that the already defined second is just the composition of first and rest; with repeated compositions we may extract any specific element from a list: e.g.

```
fifth = first:rest:rest:rest:rest .
```

Just as it is useful[4] to be able to test whether a file is empty, it is good to have the predicate, null? for testing whether its argument list is empty, or null:

```
null?:⟨⟩    = TRUE ;
null?:⟨8 4⟩ = FALSE .
```

Atom? is a predicate that tests whether its argument is an elementary (atomic) type, like integer or boolean, or whether it is a list and divisible using first and rest:

```
atom?:8     = TRUE ;
atom?:⟨8 4⟩ = FALSE .
```

Finally, the binary function cons is used to construct new lists; its two arguments are, respectively, the the first and the rest of the new list-to-be. Because it always returns a list as a result, one might argue that it is n-ate, and this perspective will become useful in Section 9.[5]

---

[4] but not necessary. Even the first FORTRANs could not test empty files.

[5] Although some might claim that it is binate---from the perspective of first and rest--- this is incorrect.

```
cons:⟨4 ⟨⟩⟩ = ⟨4⟩                    = (4 ! ⟨⟩)                ,
cons:⟨8 ⟨4⟩⟩          = ⟨8 4⟩                  = (8 ! ⟨4⟩)          ,
cons:⟨16 ⟨8 4⟩⟩       = ⟨16 8 4⟩  = (16 ! (8 ! (4 ! ⟨⟩))) .
```

At the right of these equations I introduce an exclamation point as an infix notation for cons; it should not be surprising that, like plus, cons is so important a binary, unate operator that it has been granted a shorthand, infix alias! Thus, the grammar is extended, and in two ways while we're at it:

```
E  ::=     (E ! E)  |  ⟨E *⟩      .
```

This use of asterisk (suggestive of Kleene's star) denotes an infinite list homogeneously composed of a single value. For instance, we might define a zero-vector as a solution to this equation:

```
⟨0 *⟩      = zerovec   =          (0 ! zerovec)          .
```

Or the meaning can be explained with three axioms:

```
first:⟨a *⟩ = a                    ,
 rest:⟨a *⟩ = ⟨a *⟩     ,
null?:⟨a *⟩ = FALSE = atom?:⟨a *⟩ .
```

Such "infinite" lists will be useful with functional combination, to be defined later.

The observant reader will notice that all uses of angle brackets may also be perceived as shorthand for applications of cons; even the asterisk used with angle brackets may be avoided by solving equations like the latter one for zerovec.

Let us close this section with an example of list construction that should be distinguished from cons, whose arguments are to be perceived as "element" and "suffix" of the resulting list. Append3 takes three lists as arguments and returns the list that is their concatenation. In this case the arguments are homogeneous.

```
append3:⟨a b c⟩          =: if:⟨     if:⟨null?:a null?:b FALSE⟩  c
                          null?:a    (first:b ! append3:⟨a rest:b c⟩)
                                     (first:a ! append3:⟨rest:a b c⟩) ⟩ .

append3:⟨⟨1 2 3⟩⟨4 5 6⟩⟨7 8 9⟩⟩      = ⟨1 2 3 4 5 6 7 8 9⟩  .
```

The first line of the previous definition will scan much more easily if the nested conditional is recognized as a conjunction; the predicate and will be defined in the next section.

7. Lazy Lists

The next point deals with real-life computer implementation; it is one of confusion for trained programmers rather than for laymen because what follows differs from conventions of all common language. In a sense it is merely a detail of implementation that extends a language, but it becomes necessary here because I really do want every non-unary function to take a list of arguments ---even the conditional primitive, if.

The point is that nothing is required about evaluation order within lists. To a mathematician, this lack of restriction means that evaluation must be as general as possible, so I shall describe the most general evaluation order: don't evaluate list elements until they are accessed. That is (and in contrast to FORTRAN, COBOL, PASCAL, etc. that force evaluation of contents of a data structure before the structure, itself, "exists"), evaluation of elements is postponed as long as possible - until their values really becomes critical to the computation.

Consider the conditional expression:

if : ⟨ (1 = 1) 7 undefined ⟩.

Undefined means some uncomputable value, like the solution to the equation

x = (x + 1),

the argument to if is a list of three values, but evaluation of the whole expression (to seven) only requires that two of them be evaluated. You may assume that evaluation of undefined leads to a computation that never stops (which is about as undefined as one would want), and mandatory evaluation of every argument in a list, therefore, would preclude us from finding seven as the result of this simple example.

As a parameter-passing mechanism, this convention is variously known as "call-by-name" or "call-by-need."[6] It suggests that argument or list-element evaluation is postponed until a particular element is really needed to determine the course of computation.

Again, unfortunately, using arithmetic expressions as our model for applicative programming leads us away from this convention. In most instances arithmetic primitives are strict. That is, in addition to being binary and binate, arithmetic operators are perceived to require evaluation of both operands. A sum depends on all addends; a difference on both minued and subtrahend. A product usually depends on multiplier and multiplicand, the exception being the rare multiplication by zero. How different is the situation with the logical operations of conjunction and disjunction, where anding a false factor, or oring a true term determines the result without evaluation of the other argument. This observation is useful for half the uses of these functions. (if the operands take random values; three-quarters, if arguments can also be evaluated simultaneously.)

We might define dyadic functions and, or as follows:

```
or:⟨a b⟩    =: if : ⟨a TRUE b ⟩.
and:⟨a b⟩   =: if : ⟨a b FALSE⟩.
```

If the first argument to and and (exclusive) or were some random truth value (say, that a coin flip comes up "heads") then the second argument

---

[6]There would be an operational distinction between these terms (as well as a semantic difference) if assignment statements were also in the language. I use the latter because it has the intended meaning in both contexts.

is only evaluated half of the time.

The example of and raises another point about function definition: sometimes we should like to allow the number of arguments to be arbitrary. That is, we would like to allow and to take an arbitrary number of conjuncts, actually evaluating only the minimal prefix of the list of its conjuncts. There are two styles for presenting the definition:

```
and:conjuncts          =: if:<    null?:conjuncts  TRUE
                                  first:conjuncts  and:rest:conjuncts
                                  FALSE>.

and:<>                 =: TRUE  ;
and:(a ! suffix)       =: if:<    a if:suffix
                                  FALSE>.
```

The reader might try to define inclusive-, and then exclusive- or similarly.

The first definition above is closer to LISP style and closely fits the primitives defined in this article. The second is closer to Prolog-style and denotes the same algorithm; a case analysis of the argument structure is implicit in the second which appears explicitly (as the first test) in the first. The latter form, however, may avoid repeated and conceptually redundant uses of first and rest, as the next example illustrates.

Suppose that the only conditional expression available were if2, defined exactly as if was defined above, except that it required an argument list of length precisely three: a truth-value and only two "conditional" values. (The conditional expression of ALGOL 60 coincides with if2.) How could if be defined from if2?

```
if:condpairs                   =: if2:<   null?:rest:condpairs first:condpairs
                                  if2:<first:condpairs second:condpairs
                                       if:rest:rest:condpairs          >

if:<elsepart>                  =: elsepart  ;
if:(truth ! (value ! suffix))  =: if2:<   truth value
                                  if:suffix>.
```

Again, these two definitions are intended to coincide except for syntax, with the ability to name fields within data structures making the second a bit more readable. This definition also illustrates the fact that lists need not be homogeneous. That is, the condpairs list contains both truth values and candidate values for the expression, itself; only the (alternating) access pattern distinguishes one value from the other.

8. Streams

This section is short, but terribly important. It deals with Input/Output, which has been the weak point in program language design from the very beginning. Because I/O depends so much on the implementation environment, specifying it without redefining the surrounding system is nearly impossible; the ALGOL 60 definition said the least about it: nothing.

We now have, in the primitive cons, the ability to construct either list-like or tree-like structures. Files are best percieved as if they were built in the same way---by the same primitive. Most often we deal with sequential files as if they were lists of characters.

Peter Landin long ago proposed the concept of a stream, a list that exists (is available for manipulation by first, rest, null? , etc.) even though its suffix may not exist. A fine example of this is the stream that travels down the wire from a keyboard into a computer; its suffix "exists" at no point in time, yet---conceptually--- the stream exists over time. In that sense an operating system may bind it to a filename and manipulate it as an integrated object.

The idea of "lazy lists" provides us this sort of file directly. All we can do with such a (non-empty) list/file is to extract the first (character) and to compute with the rest. The usual I/O primitives in conventional languages allow us no more.

Random access files can be built from the ability of cons to represent trees. Circular structures and file directories are possible, but their elaboration would entangle us in a discussion of "naming," a generalization of the argument/parameter naming implicit in function application.

An infinite list of numbers behaves much like a file generated from some machine, so I shall, instead, offer a simple view of sequential files through that model.

9. Lists as Functions

Consider now the problem of computing the list of Fibonacci numbers:

⟨1 2 3 5 8 13 21 34 .....) .

This list is characterized by each element being the sum of its two immediately predecessors. By way of motivation, I'll use a problem confronted by Samuel F. B. Morse:

> Given two alternative signals (. and _) that are respectively of length 1 and 2 (including the pause after the . or _), how many different codes can be formed with signal strings of various lengths? For example, there are
>
> 1 of length one . ≅ e ;
> 2 of length two .. ≅ i ; _ ≅ t ;
> 3 of length three: ... ≅ s ; _. ≅ n ; ._ ≅ a ;
> 5 of length four: .... ≅ h ; _.. ≅ d ; ._. ≅ r ; .._ ≅ u ; __ ≅ m .
>
> Morse's problem is to extend this sequence.

The result is the well-known Fibonacci sequence[7].

---

[7] An excellent overview is Section 1.2.8 of Knuth's The Art of Computer Programming (Vol. 1).

Our problem is to write a program that would generate this sequence (at least as far as the capacity of our computer's adder permits). Under the rules of the preceding section, that an element of this list is not computed until it is accessed---say, by the printer; it is not too hard to understand the following program:

```
fib:⟨i j⟩   =: (j   !   fib:⟨j   (i + j)⟩ ) .
```

where the solution we seek is given by

```
fibonacci   = fib:⟨1 1⟩ =  ⟨1 2 3 5 8 13 .... ⟩  .
```

Because of the convention that lists' contents are not computed and suffices aren't unfolded until accessed, however, the top result is quickly available:

```
(someth1ng ! suff1x)
```

where "someth1ng" will become 1 and "somef1x" will become

```
(someth2ng ! suff2x)
```

upon access.

Actually the sequence we want is merely the vector sum of two accessible lists:

```
fibonacci   =  (1 ! (2 ! vectorsum::⟨fibonacci rest:fibonacci⟩ )) .
```

Each of the lists may only have its first element accessible, but that is sufficient to unfold rest of it incrementally. (Such recursive use of lazy cons is the exception to the need for conditional expressions within recursive definitions, as used in Section 5.3.)

Vectorsum may be perceived as

```
vectorsum   =  ⟨plus *⟩
```

under the following convention, known as functional combination.[8] A list as a function requires a list of (sub)list-arguments; Perceived as a matrix represented as a list of rows (each a sublist), each function in the function list is applied to a list of arguments corresponding to the appropriate column of that matrix; the list of results is the result of the functional combination. For example,

---

[8]Those familiar with MAPping functions in LISP, especially MAPCAR, or "apply-to-all" convention in FP will recognize their generalization in the examples of functional combination that follow.

```
⟨plus minus times divide⟩:⟨
⟨ 2     4     6     8   ⟩
⟨ 1     3     5     7   ⟩ ⟩   = ⟨ 1    1    30   ⟨1 1⟩ ⟩ ;

⟨plus minus times divide⟩:⟨
⟨ 2     4     6     8   ⟩
⟨     2          *      ⟩ ⟩   = ⟨ 4    2    12   ⟨4 0⟩ ⟩ ;

⟨   minus        *      ⟩:⟨
⟨ 2     4     6     8   ⟩
⟨     2          *      ⟩ ⟩   = ⟨ 0    2    4    6 ⟩ .
```

It is often useful to write functional combination out aligned vertically, as above, to clarify the columnar application convention.

The convention of using asterisk for forming infinite lists combined with functional combination is particularly powerful when used with functional combination. Several matrix operations are surprisingly easy to program: if

```
identity:a  = a  ,
```

what is the result of evaluating

```
⟨   identity        *    ⟩:⟨
⟨ 2     4     6     8   ⟩
⟨ 1     3     5     7   ⟩ ⟩   ?
```

What would a linear algebraist call the function ⟨identity *⟩ ? How would

```
sigma:addends              =: if:⟨     null?:addends 0
            (first:addends + sigma:rest:addends) ⟩
```

help in defining the function dotproduct?

10. Why Functional Programming?

The preceding sections merely introduced a peculiar style of programmming. Experienced programmers might well ask, "What is the point?" Why do some seriously propose this radical discipline of programming when there is already so much software in the field written in what I call imperative programming, and written quite successfully in light of the expansion of the scope and accessibility of computing over the past ten, twenty, or thirty years. Why fight success?

There are two answers to such questions. Both are trends that have long been present in computer science, but that have been unified and made tractable by the recognition of functional style. I touched on the first earlier: for some time there has been an effort to incorporate the rigor of mathematics, especially algebraic transformations, into the manipulation of programs; functional style facilitates such the algebra of programs because it is so close to the familiar form and algebra of mathematical equations.

The second is the problem of programming for a richly parallel execution environment. Novice programmers learn "programming" in a language that has been derived from FORTRAN, whose arithmetic formulae

were patterned after mathematics (a revolutionary idea) but whose control structures were modeled on the Von Neumann uniprocessor computer of the 1950's. Now that microcircuit technology has made processors so fast that improvements are constrained by absolute, physical limits, and so cheap that the memory of a typical computer is orders of magnitude more expensive than the processor (a drastic reversal from the fifties), it now makes economic sense to swarm many processors around data occupying precious memory. The problem is that traditional FORTRANesque languages cannot handle the problems of parallel control for such a machine.

10.1. Algebra

We ought to be manipulating programs with the same facility that we manipulate algebraic equations. Initially FORTRAN was a tremendous success because it extended the notation of classic mathematics into the programming task. Whether we should adopt the language of mathematics *in toto* for programming is doubtful. (It is not clear what *the* language of mathematics is, anyway!) We should, however, carefully consider any way of incorporating classical mathematical syntax and semantics into a programming notation. Where the notation is suitable, we would be foolish to cast aside so many years of refinements and so many hours of learning.

This manipulation facility is important in light of the various demands placed on the programmer, and on the language he is using. Some programs *must* be correct. Validation, in its extreme--proving code correct, is often the hardest part of such programming tasks. The example of Section 5.5 illustrates the facility available when that language is close to universally-known notation. I would not have attempted that proof in an imperative language for this audience[9]; the universally accepted and timeless notation of equations would not have been so readily available. With such tools in hand, however, verification and proof becomes much like algebraic manipulation, involving distributive and associative laws. These techniques extend to surprisingly complicated programs, as we shall see in Section 11.1.

The implementation of a program that has been proven correct is only as correct as the implementation of the language in which it is expressed. Developing formalisms for semantics all share the underlying perception of a program as a mathematical function from inputs to outputs. The ground rules for writing functional programs, therefore, are most similar to those for writing formal semantics. As a result, it is quite easy to join a program's definition to a definition of the language in which it is expressed, obtaining a deep definition for a program subject to aforementioned proof techniques. Indeed, this similarity of styles often prompts breadboarding of new languages or language features in a functional language (notably LISP) as part of their development.

---

[9]Either I would have been forced to introduce one of the notations that have been developed for such proofs, or I might have attempted an English proof saturated with strained, temporal subjunctives.

This nearness to the style of formal semantics also makes it easier to specify and to implement correct program transformations, such as compilers and program optimizers. That is, the process of messaging code into another form, generally one that allows it to be executed more efficiently, also proceeds along lines of algebraic laws derived from the semantics (set of axioms) that defines the language. While compilers have been designed and built for years without all the rigor of formal semantics, current research in language semantics is paralleled by discoveries that explain how a semantics is carried through the implementation of a compiler. It turns out that conventional techniques do have formal support.

Finally, we should eventually see an improvement in the facility with which a program, expressed functionally in a language under a rigorous semantics, is maintained and revised. There is now little experience to support such a claim, but a program that reads so much like its own specification or proof should be more easily understood by someone more comfortable with any of algorithms, specifications, or proofs. Moreover, functional definitions seem to be limited to a few lines by the need to establish new parameter bindings, by the limitations of an extended conditional expression, or by the natural need to subdivide a function into named pieces after it exceeds a certain threshold of intellectual content. This inherent modularity of functional definitions implicitly creates a remarkable testing environment that is easily turned into a maintenance environment, when the maintenance engineer tinkers with pieces of an extant program just as the author tested each function-piece when he originally wrote it.

One feature often associated with readability and efficiency is the requirement to declare types of various identifiers. Rigorous type declaration is desirable but function-valued identifiers must be declared, as well. Robin Milner and his associates at the University of Edinburgh have developed a preprocessor that derives type information from the primitive operands and operators deep within the program, and propagates it up through levels of functional definitions. Just as run-time storage management (generally "garbage collection") is often left to the system, the validation of type correctness may be left to the compiler. ML, the Edinburgh language which admits higher-order functions (i.e. with functional arguments and results), can detect type inconsistencies between formal parameters and actual arguments without requiring the programmer to declare anything. (Therefore, I have not used explicit typing in the examples here.)

10.2. Parallelism

A major motivation for the interest in functional programming is the realization that we can now afford to build "supercomputers," machines with many, many processors——if only we knew how to program them. For quite some time various parallel architectures have been available and there has been much effort invested in tailoring certain algorithms to take advantage of the variously enriched efficiencies available. The experience has been that a few tailored algorithms run extremely well, but that the multiprocessor resource remains idle most of the time while the system behaves like a uniprocessor, synchronized

by the control structure.

We would like to revamp control structure so that the system avoids uniprocessor mode as much as possible, making efficient use of the multiprocessor resource, once the price of transmitting a problem into main memory (usually serially) has been paid. In particular, we would prefer to avoid swapping, in favor of a sustained and massively parallel computation.

Functional programming offers that opportunity. Without side effects, there is no possible conflict among processes. There are many opportunities for parallelism, moreover, that can be extracted from the mathematical notation (which exhibits no implication of time or sequentiality). The notation introduced above, for example, allows parallelism in evaluating the several elements in a list simultaneously. (In Section 7 we observed that these elements need not be evaluated until after they were accessed; now we observe that they may also be computed any time before that if the resource is available.) By implication we may evaluate all arguments (in an argument list), all columnar results from functional combination, and contents of (arbitrarily deep) sublists in parallel. In fact, the ubiquity of list structures assures that there are plenty of processes available to occupy idle processors; with a program implicitly fragmented into non-contending processes there is no need to "choreograph" processes to avoid conflicts. (There remains the problem of how to schedule processes onto processors.)

Like storage management, however, parallelism is not part of the denotational semantics of a language and may remain invisible except to the expert programmer; however, it might well be part of an implementation that takes advantage of any (sufficiently weakly defined) semantics. In composing the functions in this paper I was not conscious of execution order; I concentrated on meaning---correctness, alone. In this way parallelism is left to the execution environment, and the same program might run well on several parallel architectures without any detailed tailoring of code. Of course, if the architecture and the quirks of a peculiar scheduler are known, then some sort of program annotations might enhance performance, but these should be optional, certainly independent of the "denotation" of the program.

A clue to solving the scheduling problem comes from the observation that process initiation and termination will always generate undesirable processing overhead. If processes are only initiated near the leaves of the process tree, then too often these will terminate quickly, increasing their relative overhead. Better to dispatch processes near the root of the process tree, processes that will run for some time (and create needed results) before they stop. In a list-oriented functional language it is easy to identify, say, three such independent processes: they compute the next three items on the stream being printed (Section 8). In an iterative language, like FORTRAN, it is more difficult to identify such processes, because the root structure of the program is a sequence of interdependent statements to be performed in an order; while we find parallelism within FORTRAN's arithmetic expressions, they reside close to the leavers of the processing tree and will not likely occupy processors long enough to recover dispatch overhead.

A final impact that the functional programming style already offers for parallelism is an expressive facility, independent of confining uniprocessor languages, for refining good algorithms and for discovering new ones in anticipation of (and to help design) future multiprocessors. Because the constraints of every programming language shapes the horizons of its practitioners, and because almost all programmers are best trained in uniprocessor languages, we are not well equipped to discover superlative algorithms for general multiprocessing architectures. While there is a good deal of work on fitting a specific architecture with a specific algorithm for solving a particular problem, there is no comparable effort for developing algorithms whose performance advantages only manifest themselves under massive parallelism. Functional programming offers a convenient method for developing and comparing general algorithms of this sort, independently of particular computer models.

11. Algorithms

As a rule of thumb, the better algorithms for solving particular problems are recursive, rather than iterative. That is, the power of purely applicative programming is more likely to suggest excellent algorithms than iterative style. While I know of particular cases where this rule may not hold,[10] it certainly seems that recursive and functional thinking is a tool more powerful than iterative and procedural thinking.

In this section I support this observation with two examples of "good" algorithms expressed in the language developed above. Perhaps it will encourage readers to invent new ones, particularly algorithms that may not perform as well on existing architectures (probably because of overhead of control structures to carry out all implied processes), but which may shine in a parallel environment.

11.1. Sorting

Every student of iterative program has seen a couple solutions to the problem of sorting a list of numbers. Usually the solution is some kind of interchange sort, which seems easiest in the iterative language being learned (here, in PASCAL).[11]

---

[10] It's hard to be certain where new discoveries are possible. These known algorithms are highly serialized, mixing side-effects from iterative style into applicative style.

[11] As I was editing this section, Jon L. Bentley considered this very problem in his montly column, "Programming Pearls," in Comm. ACM 27, 4 (April, 1984), 287-291. I recommend his article to those who would probe deeper into this problem.

```
TYPE vectyr = ARRAY[1..N] OF INTEGER;
PROCEDURE  sort(VAR vector:vectyr);
VAR i,j: 1..N;
BEGIN
    FOR i:=1 TO N-1 DO
       FOR J:= i+1 TO N DO
         IF  vector[i] < vector [j] THEN {already ascending}
         ELSE exchange(vector[i],vector[j])
END
```

There are many variants on this nested-loop solution to sorting: insertion sort, bubble sort, selection sort, etc., but all share the property that, in the worst case, nearly $N^2/2$ comparisons are made.[12]

I have attempted a translation of this algorithm into the applicative language described above. As you read it you should be aware of a useful convention: when a function is specified to return a heterogeneous result composed of, say, two subresults, its name is a hyphenation of the two names of the subresults. Here the function min-residue returns two subresults, the smallest integer in the vector, and the vector, slightly scrambled and with that integer removed; it corresponds to the inner loop in the PASCAL code above.

```
sort:<vector>              =: if:<    null?:vector     vector
                           cons: <first sort>:                    <
                                 min_residue:<first:vector rest:vector> > > ;

min_residue:<candidate unseen1>=: if:<   null?:unseen1    <candidate unseen1>
                           less?:<candidate first:unseen1>
                                    <second    cons    >:            <
                                    < <> first:unseen1>
                                    min_residue:<candidate rest:unseen1> >
                           <second   cons    >:                  <
                           < <>   candidate >
                           min_residue:<first:unseen1 rest:unseen1> >   > .
```

Functional combination plays a significant role in this code. The columns suggest that the min subresult is always that returned by the recursive invocation of min-residue, and something is always added to the residue from that recursive call.

This exercise of translating from an iterative program to an applicative program is strained because information (about the array structure and in-place management of vector storage) is actually removed. In practice the translation should go the other way, with information dependent on a particular implementation environment (here, storage management) being added.

I find the applicative version of sort above, strained and awkward (relative to applicative style) where the PASCAL code seems deceptively

---

[12]The notation $O(N^2)$ is usually used, suggesting that the worst-case number of comparisons grows proportionally to the square of the problem size, N.

clean (relative to its style). While it is possible to express the same algorithm as we just saw in PASCAL, it is not easy to. When I first did this back-translation exercise, I struggled to remain faithful to the PASCAL code because intuition screamed that more useful results could be returned from the machinery of the min-residue function. The vector traversal there could as well partition the list, instead of just returning the minimum and the list's remainder. Following that temptation, one would rediscover a much better algorithm due to C.A.R. Hoare, called "quicksort", which is derived with but a slight alteration in the functional combination in qsort, below. Here the analog of min-residue is lt-eq-gt, which reshapes its argument list into three vectors of integers less than, equal to, and greater than the boundary value.

```
qsort:<vector>          =: if:<     null?:vector      vector
                        append3:    <qsort first qsort>:                <
                                    lt_eq_gt:<first:vector rest:vector> > > ;

lt_eq_gt:<boundary list>      =: if:< null?:list      <<> <boundary> <>>
                        less?:<first:list boundary>
                                    <cons second second>:         <
                                    <  first:list   *  >
                                    lt_eq_gt:<boundary rest:list> >
                        greater?:<first:list boundary>
                                    <second second cons>:         <
                                    <  first:list   *  >
                                    lt_eq_gt:<boundary rest:list> >
                        <second cons second>:         <
                        <  first:list   *  >
                        lt_eq_gt:<boundary rest:list> >  > .
```

The auxiliary functions append3 and second have all been defined previously.

Much can be learned from studying these two algorithms. The worst-case behavior of either one requires time (number of comparisons) proportional to N^2. Average time of quicksort, however, is proportional to N(log N) where there is no difference between worst and average case with sort. The improved performance occurs in the ideal case that lt-eq-gt divides the list in half or thirds; the worst case performance occurs when it behaves just as min-residue did, with all other elements either above or below the boundary value, when one subresult is all but as long as the original argument.

While sort may actually be better than qsort for sorting very small vectors, qsort will be better for a problem of any significant size.[13] This is an excellent example of the "divide and conquer" paradigm of programming. The efficiency of qsort is due to the likelihood that the linear traversal of lt-eq-gt will cleave, rather than peel, the original problem into subproblems.

---

[13] See the discussion of hybrid algorithms in the following subsection.

Qsort is only the simplest representative of a family of sorting algorithms, known as "partition sorts," that attain asymptotic optimality: expected sorting time proportional to input size. I mention them because they can be so easily characterized in this context:

```
dpsort:vector          =: if:<    homogeneous?:vector    vector
                          appendn:<    dpsort   *  >:  <
                                       partition:vector  > > .
```

The auxiliary function appendn is append3 rewritten to take an arbitrary number of arguments, much like and above. The functions homogeneous? and partition return results befitting their names, but usually are performed with a single function, homogeneous?_partition, that determines whether each partition is homogeneous as it is built; for larger vectors, partitioning into more pieces yields better performance.

The applicative code, above, is only "source code" and is subject to much algebraic manipulation. For instance, the similarity of the alternative result-expressions in lt-eq-gt suggests that we apply a distributive law (of conditional over function application) to get

```
lt_eq_gt:<boundary list>      =: if:< null?:list      <<> <boundary> <>>
      (if:< less?:<first:list boundary>   <cons second second>
            greater?:<first:list boundary>     <second second cons>
                                               <second cons second> >):      <
                                               < first:list   *  >
                                               lt_eq_gt:<boundary rest:list> > > .
```

One more distributive law (of conditional over list formation) yields a single function-list like

```
<if:<lt cons second>  if:<eq cons second>  if:<gt cons second>>
```

where the identifiers lt, eq and gt are the three boolean subresults of a single comparison of boundary and first:list.

That is to say, this code is still subject to much manipulation before its performance on specific data might be measured. Such manipulation might take the form of translation (into PASCAL or into a "machine code"), something that is generally accepted from all programming languages. But it also might include determination of data structures (linked or sequential allocation), static (in place) or dynamic (virtual memory) space resources, or storage management (garbage collection, hardware reference counting, or provided ''for free' by an overseeing operating system.) Certainly the implementation need not be faithful to the apparent recursion pattern if algebraic manipulation allows another interpretation.

Most importantly, consider implementation of these sorting algorithms in a brilliantly multiprocessing environment. The applicative code specifies no assignments of values to "specific" variables; thus there is no implicit synchronization. The partitioning implicit in qsort really shines in this light. A large sorting problem is immediately decomposed into two (nearly equal-sized, we hope) subproblems that are mutually independent. Each will occupy a processor for some time and is, in turn, subject to further decomposition as long as idle processors are available and the subproblems are significantly large. In

this way the overhead of processor dispatch and recovery can be distributed over much useful processing.

(In such an environment sort generates dependent subprocesses, which exhibit a "cascading" behavior under lazy evaluation.) Thus, as we have two, four, eight, etc. processors, the qsort code (or the sort code, for that matter) need not be retailored to run well on each machine.

11.2. Matrix Representation

The material in this section is offered to demonstrate the power of programming applicatively, both in terms of uncovering good algorithms and also for crisply expressing parallelism. Readers not yet comfortable with applicative style or unfamiliar with linear algebra might want to skip this on first reading. In presenting these algorithms I am make no comparisons with other programming styles, except to observe that these programs cannot be cleanly written any other way. They offer dramatic improvements over those common use. They are transparent to identification of (mutually independent) parallel processes, res ipse loquitur in different ways to different readers.

Let us represent a matrix using recursive lists. Somewhere (not necessarily local to the list) there will be need for some sort of "header" information (e.g. dimension, scalar representation, number of rows, columns, etc.) which I shall not specify precisely. The crux is the list structure itself: a matrix is defined to be one of

- -the empty list (representing an all-zero matrix);
- -an integer or scalar (representing a 1x1 matrix);
- -a list of four matrices (the quadrants explained below).

In the last case we envision a matrix as cleaved, once vertically and once horizontally, into quarters, each exactly one-fourth the size of the original matrix. The quarters, or quadrants, are indexed as one reads English: the first is upper-left and the third is lower-left. By inference all matrices are square and sized by the largest power-of-2 not exceeding both the height and width of the "intended" matrix.

How can square matrices of size 1, 2, 4, 8, etc. efficiently represent all other sizes? The answer rests in the use of the all-zero matrix. What we precieve as a 12-by-24 matrix will be represented as a 32-by-32 matrix, whose third and fourth quadrants are null. Its second quadrant will similarly have two null sub-quadrants, and so forth; see Figure 1. Although Figure 1 appears to have capacity for a much large matrix, not very much space is wasted in representing non-existent (i.e. zeroed) rows and columns.

At first one suspects that the only quadrant that is really "full" is the first, but even this assumption may not hold. A covey of zeroes in the upper left corner of a matrix will also be represented as empty quadrants (nulls), with the desirable result that the representation takes far less space than conventional representations. How would a 33-by-33 matrix containing all zeros, except for a 1 in the dead center, be represented?

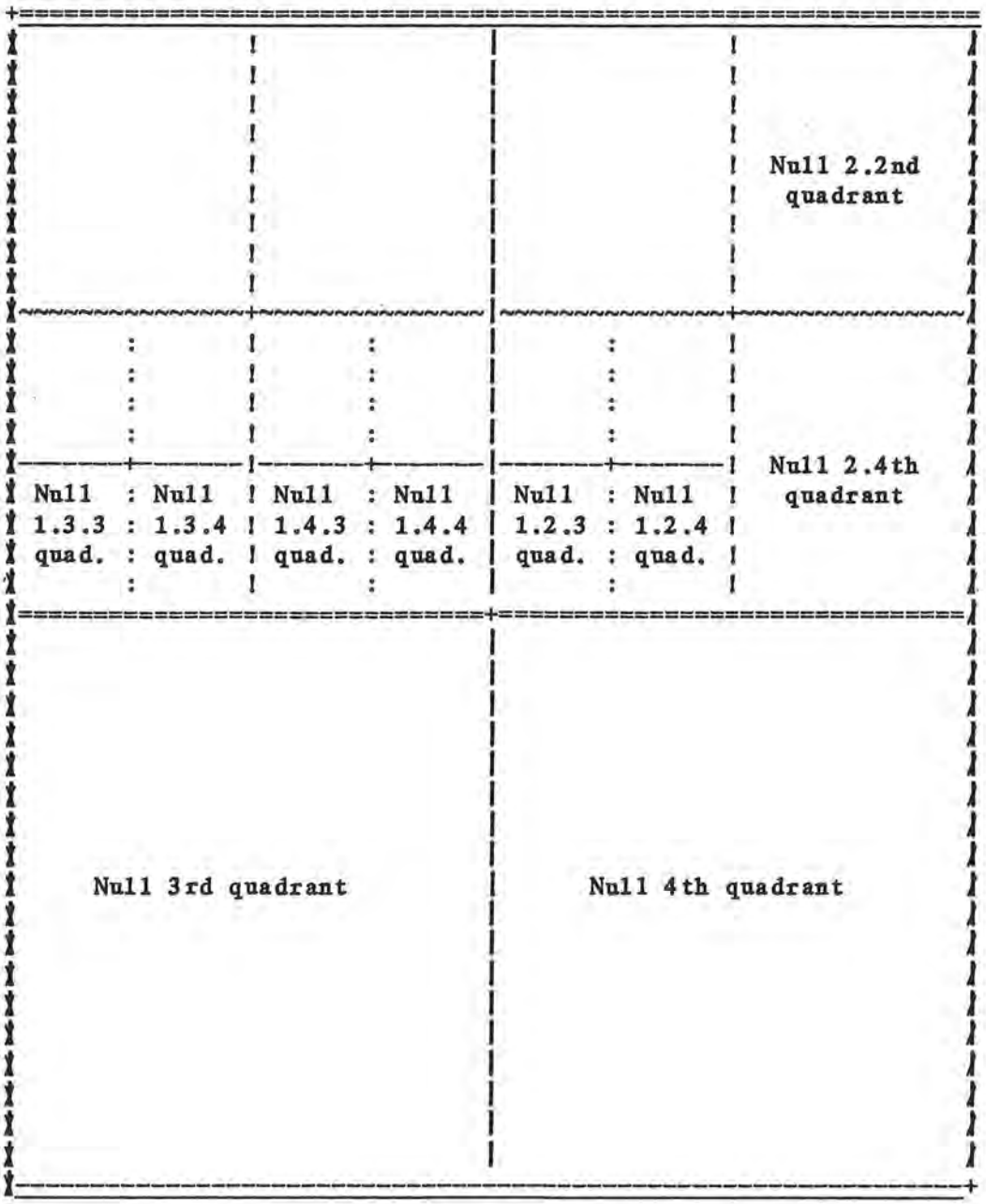

Figure 1. A 12 x 24 matrix in 32 x 32 quad representation.

Space-efficient representations for sparse matrices have been known for some time, and some have even been incorporated in vector-machine hardware. While the representation proposed here is not space-optimal, it does compress matrices in a way that lends itself to matrix operations, as we shall see. For now, observe that on null (zero) matrices addition, multiplication, transposing, and testing-for-singularity are all trivial operationally, as well as conceptually.

The natural analog for representing a vector is a binary tree, for representing an n-dimensional array, a 2^nary tree. This vector representation should be contrasted with the list representation from Section 8. There I used an asterisk notation to imply distribution of a function over a linear list; it or a a similar notation might also indicate distribution over a tree. As we shall see below, some languages (FP and APL) specify little or nothing about the internal representation of vectors, and so are free to provide explicit or implicit functional notation for distribution of functions over trees.

In the following algorithms I have made no provision for size information that might be present in the matrix header. Such information is not essential here.[14] As a result, my code has more tests than would be necessary if size specification is verified beforehand; a counter would be used (instead of the atom? tests here) to curtail recursion. Detection of size conformability could thereby be handled before, not during, the application of these particular functions. (It is easier to assume here that all matrices are square and that all operands conform.)

Probing a specific entry within a matrix, however, does require knowledge (from the header or other specification) of the size of the matrix, that is, the maximum of its length and width. Let us define a function to access the i,jth element of a matrix whose largest side is, at most, twice the integer middle; For Figure 1 the value of middle would be 16.

```
access:<matrix middle i j> =: if:<
      null?:matrix 0
      and:<(middle = 0) atom?:matrix (i = 1) (j = 1)> matrix
      or :<(middle = 0) atom?:matrix (i = 1) (j = 1)> conformerror:<matrix middle i .
      if:<(i < middle)
            if:<(j < middle)
                  access:<  I:matrix half:middle    i           j         >
                  access:< II:matrix half:middle    i       (j - middle)> >
            if:<(j < middle)
                  access:<III:matrix half:middle (i - middle)   j         >
                  access:< IV:matrix half:middle (i - middle)(j - middle)> >>>
```

Some easily defined helper functions are assumed above; in particular, I, II, III, and IV access the appropriate quadrants:

```
 I:mat    =: first:mat                    ;
II:mat    =: first:rest:mat               ;
III:mat   =: first:rest:rest:mat          ;
IV:mat    =: first:rest:rest:rest:mat     ;
half:i    =: first:divide:<i 2>           .
```

Access of a specific element takes time logarithmic in the size of the matrix, or O(log(size)).

---

[14] On the contrary, these algorithms do not need rigorous size checking, running correctly whenever sizes conform within an order of magnitude (taken as the power of 2).

For the purposes of input and output, we should like to present matrices in either row-major or column-major form. Now that the quadrant-representation is understood, the reader is encouraged to define translation functions between row-major presentation of matrices (of Section 8) and this "quad-tree" form.[15] The program is not easy, but fortunately it is executed relatively rarely.

Section 8 reminds me of the problem of matrix transpose. We can effect the transpose of a matrix represented in this way merely by exchanging the definitions of II and III throughout. Rather than a partial recopying or a complete rebuilding of an extant matrix, all we need to do is to arrange a rebinding of two probing functions:

```
II:mat =: first:rest:rest:mat          ,
III:mat =: first:rest:mat              ,
```

by a boolean in the matrix header, which selects how II and III are to be interpreted for that matrix. Transposing a matrix is effected by just providing a new header (which would be necessary anyway) for the untouched list representation.

Matrix addition is easy for "untransformed" matrices. The only trick is to maintain the normal form when submatrices sum to zero.

```
matplus:⟨a b⟩          =: if:⟨    null?:a  b
            null?:b  a
            and:⟨atom?:a atom?:b⟩  scalar:(a + b)
            or :⟨atom?:a atom?:b⟩  conformerror:⟨a b⟩
            normform:⟨matplus *⟩:⟨
                                a
                                b   ⟩     ⟩     ,
scalar:i    =: if:⟨    (i = 0) ⟨⟩ i ⟩          ,
normform:quad          =: if:⟨ and:⟨null? *⟩:quad ⟨⟩ quad ⟩     .
```

The time to add two matrices is, again, at least O(log(size)), even when there are n^2 adders in parallel; the logarithmic time arises from the quad structure, just as in access.

The preceding code for matrix addition and that following for Gaussian matrix multiplication both exhibit a natural high-level decomposition of the matrix operation into four additions or (respectively) eight multiplications of quadrants. Functional combination and collateral argument evaluation yield the top-down decomposition that we want.

---

[15] The quad tree, which inspired these algorithms, is a well known structure for analyzing two-dimensional graphics.

```
mattimes:⟨a b⟩          =: if:⟨     null?:a  a
            null?:b  b
            and:⟨atom?:a atom?:b⟩  (a * b)
            or :⟨atom?:a atom?:b⟩  conformerror:⟨a b⟩
            ⟨matplus *⟩:⟨
                        ⟨mattimes *⟩:⟨    ⟨ I:a  I:a III:a III:a⟩
                                          ⟨ I:b II:b   I:b  II:b⟩ ⟩
                        ⟨mattimes *⟩:⟨    ⟨ II:a II:a  IV:a  IV:a⟩
                                          ⟨III:b IV:b III:b  IV:b⟩ ⟩⟩ ⟩ .
```

Running time, even with n^3 multipliers, is no better than O(log(size)).

If either of the arguments, a or b, might be "transposed" as described above, then more[16] versions of matplus and mattimes are needed, essentially with and without uniform exchange of II:a and III:a, or II:b and III:b throughout this code.

At first providing different programs for the same operations seems expensive, but it is cheaper than requiring that any matrix to be represented in either of two formats as some stream-oriented implementations do ("row major" and "column-major" form.) This "quadrant-major" representation serves both roles. ,pp When either operand is null, the efficiency of matplus or mattimes is startling. Zero quadrants are not just random, because the canonical forms for matrices exhibit such sparseness. If just one quadrant of one operand to mattimes is null, then two of its eight recursive calls and half of its four invocations of matplus become trivial, massively improving its performance. Being able to reduce eight recursions to six is a fine improvement.

In 1969 V. Strassen presented a remarkable way to multiply matrices, perfectly suited to this matrix representation, that can reduce these eight recursions to seven in all cases. It, too, is best presented in applicative style but I shall not press through it here. A fine presentation appears in Chapter 6 of The Design and Analysis of Computer Algorithms by Aho, Hopcroft, and Ullman (Addison-Wesley, 1974). Instead of the four additions that we saw above, however, it requires eighteen[17] quadrant additions, of which ten, six, and two may be done in parallel. For multiplying sparse or small matrices it is no improvement over the Gaussian multiplication, above, (just as qsort is no improvement for small sorting problems), but as matrices grow, the savings in multiplications quickly compensates for the additional additions.

[Because addition does not associate under ordinary representations of floating-point numbers, however, this quadrupling of addition operations at each step can introduce precision errors. (There is no problem with ordinary computer representations of rings like modulo-2 arithmetic or the integers, where normal algebraic laws hold.) Earlier I said that programming languages should allow us to practice algebra; here I note another restraint on that practice: number representation.]

---

[16] Actually only two for addition and three for multiplication.

[17] S. Winograd has improved this to fifteen at the cost of some synchronization.

The best algorithm is a hybrid, working roughly as follows:

-if the matrix is sized 16 x 16 or less, use Gaussian multiplication;
-if all or any quadrant of either operand is null, take a Gaussian step;
-otherwise take a Strassen step.

The Strassen or Gaussian (essentially mattimes) step reduces the problem to seven, six, or fewer matrix multiplications of this hybrid variety. The boundary size of sixteen is derived from analysis (of dense matrices), which shows this strategy to be optimal in number of scalar arithmetic operations, i.e. sum and product, and almost optimal in number of non-trivial function-invocations (which translates under parallelism to a census of process-dispatches, a critical measure of complexity there.)

The point of this section is to demonstrate the leverage of applicative style on a known problem; the code presented here not only admits more efficient algorithms, but also it remains transparent to their implementation under multiprocessing. The elegant quad-tree matrix representation yields many benefits: matrix transpose is trivial; it facilitates decomposition of other operations into large subprocesses; it allows efficient, high-level representation of sparse matrices while lending itself well either to Gaussian or to Strassen's multiplication algorithm; thereby it admits a three-way hybrid algorithms; and there are no clever exceptions hidden in its structure to hamper algorithms that we cannot now anticipate.

The last is most important, because I have not mentioned all the conventional matrix operations; Aho, Hopcroft, and Ullman discuss some. It turns out that finding inverses, determinants, eliminants (a new dual of determinants),[18] and "pivot-steps" are all operations that lend themselves to the quad-tree matrix representation.

And even more significant algorithms might remain to be discovered in this area. When such a simple observation yields algorithms this elegant, I suspect that some fundamental perspective has been uncovered, and that the underlying theory might be better recast from that angle. Surely we would be able to reproduce the known results under the reformulated definitions, but---more importantly---we also might be able to prove new results by using more powerful tools. If matrices had been originally perceived as quadrants, how could FORTRAN's EQUIVALENCE statement have been entended to matrices? And how differently would development of "vector machines" have proceeded!

12. Comparing Various Functional Languages

In this section I present the program quicksort from Section 11.1 recast into four popular applicative languages, as well as PASCAL. The choice of an example is not easy because these languages are fairly diverse. For instance. APL seems to be designed around scalar and matrix arithmetic, while LISP is built on manipulation of list

---

[18] S.K. Abdali & B. D. Saunders, Transitive closure and related semiring properties via eliminants, Theoretical Computer Science (August, 1984.)

structures. I have chosen quicksort because it mixes both problems of arithmetic (the key comparisons) and of structure manipulation. The resulting programs may not always compare, because of, for instance, conceptual differences between list and array structures.

The last section, moreover, points out constraints of overly explicit structure manipulations; there I argued that vectors and matrices should be reconsidered as tree structures. While APL seems most numeric, it also has the least constrained matrix representation; APL just may use trees internally without the user knowing. Sorting would be most efficient if the data vectors were trees, and with a bit of unseen algebra the APL in Section 12.5 could realize this optimum.

At least we are familiar with the example already. Section 11.1 uses a style that is quite close to the language, Daisy, implemented at Indiana University. It would be well to compare this with an iterative- and array-oriented language, PASCAL, before we proceed to languages less widely known.

12.1. PASCAL

PASCAL is not designed to be an applicative language. Therefore, its parameter-passing mechanism is not tuned to the style. For instance, there is a difference between the kinds and numbers of objects that can be passed as arguments and that can be returned as results; writing a binate "divide" is awkward. One does not, therefore, expect to infer parallelism from PASCAL programs.

C.A.R. Hoare invented quicksort as an in-place algorithm for languages like PASCAL. Because it "destroys" its input parameter in order to "create" its result, it violates the side-effect-free constraint on applicative languages. (For instance, lazy evaluation from Section 7 might be crippled.) On the other hand, we can expect it to achieve the high-level bifurcation desirable for multiprocessing.

The code I present in Figure 2 is a modification from Bentley's[19]; he indicates several possible enhancements.

A good compiler still has opportunity for algebraic improvements even to this highly sequential code. The comparison in the inner conditional is better distributed out of the logic to be cast as a single comparison; it might look like

```
CASE COMPARE(vec[scan], boundary) OF
      < : statement1 ;
      = : statement2 ;
      > : statement3
END
```

if PASCAL allowed such syntax.

The interesting point is that the first recursive invocation of qsort may be dispatched as a separate process. The second recursive

---

[19] from Comm. ACM 27 , 289 as previously cited.

```
CONST size   = 254;          size1  = 255;
TYPE  range  = 1..size; range1 = 0..size1;
      scalar = integer;
      vector = array[range] OF scalar;
PROCEDURE qsort(VAR vec:vector; lo,hi:range1);
VAR   boundary: scalar;
      lastequal, scan: range;
      lastlow: range1;
BEGIN
      IF lo>=hi THEN {It's already sorted} ELSE
      BEGIN
            boundary := vec[lo];
            lastequal := lo; lastlow := lo;
            FOR scan := lo+1 TO hi DO
                        {Invariant:
                          (vec[lo,,lastequal]        = boundary)    &
                          (vec[lastequal+1..lastlow] < boundary)    &
                          (vec[lastlow+1..scan-1]    > boundary)     .}
                      IF vec[scan] <boundary THEN
                      BEGIN
                            lastlow := lastlow+1;
                            swap2(vec[lastlow], vec[scan])
                      END
                      ELSE IF vec[scan] =boundary THEN
                      BEGIN
                            lastlow := lastlow+1;
                            lastequal := lastequal+1;
                            swap3(vec[lastequal], vec[lastlow], vec[scan])
                      END
                      ELSE {vec[scan] >boundary.  Let it stay there.} ;
            qsort(vec, lastlow+1, hi);
            FOR scan := lastequal DOWNTO lo DO
            BEGIN
                        swap2(vec[scan], vec[lastlow]);
                        lastlow := lastlow-1
            END;
            qsort(vec, lo, lastlow)
END   END
```

Figure 2. PASCAL code for Quicksort

call[20] then runs in parallel and asynchronously with the first recursion. The properties of the algorithm guarantees that there is no conflict between the processes.

[20]Recognizable as a so-called "tail recursion" and easily transformed into a loop.

12.2. LISP

LISP is really a family of languages traceable to one common root. There is a fairly well recognized subset that meets my definitions for a functional language, and I use a tame extension of it. If the following code is not legal in your particular version of LISP, then a fairly trivial syntactic transformation will render it so.

When reading LISP, be aware that the name of the operation being invoked occurs just to the right of the opening parenthesis, rather than just to the left of it. While some complain about the spartan syntax ((too many parentheses)), others relish it because every program is also a list; among the higher-level languages, it, alone, realizes von Neumann's unification of program and data structures.

```
(def qsort (lambda (lis) (cond
      ((null lis) lis)
      (t (let ( ((lt eq gt) (lt_eq_gt (first lis)(rest lis))) )
            (append3 (qsort lt) eq (qsort gt)) )) )))
(def lt_eq_gt (lambda (boundary lis) (cond
      ((null lis) (list lis (list boundary) lis))
      (t (let          (     ((lt eq gt) (lt_eq_gt boundary (rest lis)))
                       (this (first lis)) )
            (cond
            ((lessp this boundary)  (list (cons this lt) eq gt))
            ((greaterp this boundary)     (list lt eq (cons this gt)))
            (          t          (list lt (cons this eq) gt)) ) )) )))

(def append3 (lambda (a b c) (append a (append b c)) ))
(def first (lambda (x) (car x)))
(def rest  (lambda (x) (cdr x)))
```

We define the function qsort to take one parameter, lis and to return the result of a conditional expression. (The keyword, lambda, is a holdover from Church.) Either lis is empty, and itself is the result, or the result may be described by appending three sublists. The three are obtained by partitioning and applying qsort recursively to the first and third partition before appending.

The partitioning function lt eq gt returns a list of three lists. In the complex case that its argument, lis, is non-empty it binds this to the first item on lis; lt, eq, and gt to the partition of the rest of lis. Then its problem is reduced to consing this onto one of the three sublists, and returning the triple.

12.3. Prolog

There are two immediately noticeable differences in the Prolog code. The first is that qsort has two parameters; the second is that no explicit append function is invoked. Both have the same explanation. Prolog is less a language than a theorem prover; The user defines relations, not functions; patterns to the left of the :- symbol are equivalent to the conjunction (on the right) of patterns separated by commas.

```
qsort(Vector, Sorted) :- qs(Vector, Sorted-[]).
qs([], Vec-Vec).
qs([Boundary|Suffix], Lsort-Gtail) :-
      lt_eq_gt(Boundary, Suffix, [Lt-[], Eq-Gsort, Gt-[]]),
      qs(Lt, Lsort-Eq),
      qs(Gt, Gsort-Gtail).

lt_eq_gt(Boundary, [], [ L-L, [Boundary|E]-E, G-G]).
lt_eq_gt(Boundary, [First|Rest], [ L, [First|E]-Etail, G]) :-
      Boundary=First, !,
      lt_eq_gt(Boundary, Rest, [L, E-Etail, G]).
lt_eq_gt(Boundary, [First|Rest], [ [First|L]-Ltail, E, G]) :-
      Boundary>First, !,
      lt_eq_gt(Boundary, Rest, [L-Ltail, E, G]).
lt_eq_gt(Boundary, [First|Rest], [ L, E, [First|G]-Gtail]) :-
      Boundary<First, !,
      lt_eq_gt(Boundary, Rest, [L, E, G-Gtail]).
```

In this case the parameter, Vector will be bound to a list of integers, and the parameter, Sorted will be unspecified. In fact the latter is to be a result of a proof; Prolog attempts to validate the assertion that Vector and Sorted fulfill the specified relation, and it will find a binding for Sorted as a byproduct of the proof. That is, Vector and Sorted fulfill the qsort relation exactly when Vector and Sorted-[] meet the qs relation.

The proof is pattern-driven, and the pattern mechanism is used to force Sorted to be constructed in three pieces, already appended into one. The use of the pattern

```
Head-Tail
```

is not a subtraction; it is a reference to a sublist, referenced by its head and also by a reference to its suffix, its tail, which is not conceptually part of it. Thus

```
Vec-Vec
```

is conceptually empty for any Vec; it remains for another part of the proof to constrain the particular value of Vec.

The same pattern matching mechanism relaxes the need to structure "results" of lt_eq_gt as a list of three sublists. Here it is a relation, rather than a function, and the fact that we perceive the first as "input" and the second three as "output" is merely our perspective.

I should say a word about the two vertical strokes, "|" and "!". The first is merely infix cons, essentially as introduced in Section 6. The second is a cut and is fairly sinister. It should be perceived as a program annotation that guides Prolog toward the correct answer by cutting off uninteresting possibilities. However, its placement affects the semantics of the program; if misplaced then Prolog may not discover an intended "result" relationship. It is correct to leave out all cuts at the cost of processing time, but their correct placement requires a deeper understanding of the theorem prover.

12.4. FP

The most striking thing about Backus's language FP,[21] is that no formal parameter names appear in the definitions of functions. One should read them as if they were applied to the input (here, the vector to be sorted), with the rightmost functions operating on that input.

This absence of named objects may , at first, seem awkward but it is perfectly consistent with the functional style and the goal of "doing algebra." FP offers a very rich language for abstract manipulation of programs, including both primitive and user-defined (in FFP) "functional" operators for combining functions. Of these the most frequently used are 1, 2, 3, etc. indicating selection of the first, second, third, etc. part of a subresult.

The 'little circle' denotes ordinary functional composition. Function lists in brackets yield a result vector of the same length as the function list; each function in the list is applied to the single argument. Such an argument may be decomposed with the integer-functions.

The following code follows that above, with the exception that the first element and the entire input vector are passed to the partitioning function.

```
Def qsort = null -> []; /un@[qsort@, 2, qsort@]@@[1 ,id] .

Def lt_eq_gt =
            α  /un @ trans @ α (gt -> [[2], [], []] ;
                                lt -> [[], [], [2]] ;
                                      [[], [2], []] ) @ distl .
```

In the case of qsort, a null argument results in a null result by applying the null function to it. Otherwise, three results are united into a single list by inserting the (infix) binary, unate, associative primitive un between its elements; the functional slash, /, inserts un to effect append.

The partitioning function, lt eq gt, conceptually constructs an intermediate result that is a matrix of null or singleton lists. Each of elements in the three columns contains elements of the second element of the argument (a vector) that are, respectively, less than, equal to, or greater than the first element of the argument (an integer). (In an actual implementation this structure may never actually exist.) This matrix is transposed, and then the "append" operation, above, is performed on each row. The functional alpha behaves much like the star in functional combination (Section 9).

John Williams, who kindly helped me compose and test this code, properly points out that FP allows a much better sorting strategy. His implementation of FP has a version of the slash functional, /, that structures its argument as a (complete) binary tree. Together with a primitive merge (which does what its name implies to two sorted lists)

---

[21] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Comm. ACM 21, 8 (August, 1978), 613-641.

it allows us to write the following whiz-bang sort:

Def sort ≡ tree merge ⍤α[id] .

This code makes every element into a singleton vector (α[id]) and then builds a tree of merge's over them. Not only is this code more elegant, but also it is much faster---even without taking advantage of implicit parallelism .

12.5. APL

While the language of APL expressions is functional, much more is needed to write programs. Unfortunately, it lacks a conditional expression, forcing me to write some assignment statements and apparently serial go-to code. View that as syntactic overhead for recasting the conditional inside an expression; Lines 1-3, below, only sort the trivial vector.

APL is oriented towards arrays rather than lists, with some remarkable distributive laws. Generally, primitive operators may be applied to arrays elementwise, yielding an array of results. In the following example, I use comparisons between an integer and a vector to construct (conceptual) intermediate results that are vectors of bits, like the columns from the intermediate array in the FP code above. These are used as operands to the slash, /, operator that selects elements from that same vector to be collapsed into a shorter one. In this case no append is needed because APL's comma implicitly concatenates vectors.

```
⍀/    Z ← QUICKSORT V
[1]   → ((⍴V) > 1) / 4
[2]   Z ← V
[3]   → 0
[4]   Z ← (QUICKSORT (V<V[1]) / V) , ((V=V[1]) / V) , QUICKSORT (V>V[1]) / V

⍀/
```

Unfortunately, every operator in APL is either unary or binary, with every operand either primitive or a very regular structure. Infix notation is mandatory and no functions may be passed as arguments. This prevents me from extracting the pattern

(V relationopr V[1]) / V

from Line 4 and then abstracting relationop as a parameter to this expression. Then it could be written but once and applied to the vector of the three relational operators (suggesting but one traversal of V); as before I want V to be traversed but once during partitioning.

But APL also has the virtue that the user is kept unaware of its array representation or its pattern of execution. In fact, an implementation just might traverse V only once; it might even represent V as a complete binary tree, allowing a multiprocessing tree traversal.

Thus, the specific details that APL has denied the programmer do not constrain the implementor. An implementor of APL may choose her own matrix representation and algorithms; one could find quad trees and

recursive matrix operations already buried in an existing implementation.

13. Conclusion

This description of applicative programming is hardly comprehensive. I could have said more about the importance of passing functions as arguments/results, and functionals to use them. I only hinted at algebras for program manipulation. Nothing but numeric examples have been offered, though the reader surely understands that non-numeric programs make heavy use of lists and trees. Numeric examples were chosen as a common ground for characterizing four or five languages, some having few other applications in common.

Although I argue for applicative programming, I do not expect that the software industry will ever embrace it. Why retrain? Most of today's applications and most ordinary resources are well-suited to current imperative style.

There is a role for functional style, however. While FORTRAN and ALGOL did not eliminate assembly language programming, they did change the way that programming was taught and the way that most work was done. They did that by expanding the horizons of computer applications to problems and to people that wouldn't have gotten together in any other way. They also prompted a continuing rethinking of computer architecture. A similar future belongs to applicative programming.

For an attack on the problems that we haven't solved satisfactorily, for insight on the architectures that we haven't yet built, and for motivating the students who haven't yet learned to program, I suggest applicative programming.

Note to typesetter: For the code in the late sections on FP and APL...

$\overline{\backslash/}$ is mathematical "del".
∘ is infix little circle (function composition).
-> is right pointing arrow.
← is left pointing arrow.
α is Greek alpha.
ρ is Greek rho.