REASONING ABOUT ARRAYS

by

John C. Reynolds

July 1977

This is a preprint of a paper which has been submitted to the
Communications of the ACM.

REASONING ABOUT ARRAYS

John C. Reynolds

Syracuse University

ABSTRACT A variety of concepts, laws, and notations are presented which facilitate reasoning about arrays. The basic concepts include intervals and their partitions, functional restriction, images, pointwise extension of relations, ordering, single-point variation of functions, various euqivalence relations for array values, and concatenation. The effectiveness of these ideas is illustrated by informal descriptions of algorithms for binary search and merging, and by a short formal proof.

Work supported by National Science Foundation Grant MCS 75-22002

1. Introduction

The use of assertions to describe programs and prove their correctness[1,2,3] has developed to the point where the necessary assertions are often at least as lengthy and difficult to comprehend as the program which they describe. A major cause is the use of languages and proof methods - typically the first-order predicate calculus - which are taken from classical logic and are not oriented towards programming.

Perhaps the most glaring example of these difficulties is the use of arrays. One need only compare the assertions needed to describe a program such as n log n exponentiation, which does not involve arrays or other compound data structures, with the assertions for a program such as binary search, which is intuitively no more complex, but uses arrays. In the first case, the assertions are clear and concise, and reasoning about them involves only the familiar laws of elementary algebra. But when arrays are introduced, the assertions become lengthy and filled with quantifiers, and their manipulation seems only tenuously connected with the programmer's intuition.

Superficially, we need a better notation for assertions about arrays. But more fundamentally, we need concepts and laws which are not only correct but also reflect our intuitive understanding of arrays, just as the concepts of addition and multiplication, and the associative, commutative, and distributive laws reflect our intuitive understanding of numbers. Once the right concepts and laws have been found, it is comparatively trivial to design a notation which facilitates their application.

This paper presents a variety of concepts, laws, and notations for reasoning about arrays - some borrowed from mathematics and others original - which we believe meet the above criteria. Their utility will be demonstrated both by informal descriptions of program behavior and by a short formal proof of program correctness.

For a programming language, we will use Algol 60 with the following changes:

(1) **while** statements.

(2) Round rather than square brackets for array subscripts (which emphasizes the view that array values are functions).

(3) Integer expressions of the form **lower** X and **upper** X, denoting the minimum and maximum subscripts of a one-dimensional array X.

(Although we will not use procedures here, it should be noted that for (3) to be fully useful, there must be some way of restricting the interval of subscripts of an actual array parameter.)

We have purposely stayed close to Algol to avoid inadvertently choosing a programming language which hid the defects of our assertion language. In particular, we have refrained from introducing our notation for assertions into the programming language itself (except for **lower** and **upper**, which were irresistably attractive). Moving in this direction seems to lead to a very high level language, closer to APL than to Algol, which is beyond the scope of this paper.

On the other hand, even the choice of Algol has had subtle effects on the ensuing development. For example, switching to a programming language with the novel approach to arrays described in Chapter 11 of Reference 4 would necessitate minor changes to many concepts, such as abandoning the uniqueness of the array value with an empty domain.

To an even greater extent than is indicated by the explicit references, this work is built upon the ideas of C. A. R. Hoare.[5,6,7] Mention should also be made of distinct but related work on arrays by D. C. Cooper[8] and of work by R. Burstall[9] which, roughly speaking, does for list structures what we are trying to do for arrays.

2. Interval and Partition Diagrams

Before considering arrays themselves, we introduce some diagrammatic expressions for making assertions about subscripts. Basically, these expressions are a formalization of the diagrams which are traditionally drawn by programmers when describing arrays.

An *interval* is a finite consecutive set of integers. If a and b are expressions denoting integers, then $a\boxed{\quad b}$, called an *interval diagram*, is an expression denoting the interval

$$a\boxed{\quad b} \equiv \{i \mid a < i \leq b\} \ .$$

When formulating general properties of interval diagrams (or partition diagrams) we will always use the standard form $a\boxed{\quad b}$. But when using the diagrams to make assertions, we will permit more flexibility.

Specifically, at either end of an interval diagram, $|a$ may be written instead of $a-1|$. Also, $\boxed{a}$ may be written as an abbreviation for $\boxed{a\quad a}$. Thus

$$\boxed{a\quad b} = \{i \mid a \le i \le b\}$$
$$\boxed{a\qquad}b = \{i \mid a \le i < b\}$$
$$a\boxed{\qquad}b = \{i \mid a < i < b\}$$
$$\boxed{a} = \{a\} \quad .$$

For any finite set S, we write $\#S$ to denote the size, or number of elements in S. Thus

$$\# a\boxed{\qquad b} = \textbf{if } b - a \ge 0 \textbf{ then } b - a \textbf{ else } 0 \quad . \tag{2.1}$$

This use of a conditional expression to describe a fundamental property of a data structure is a clear symptom of a potential source of error, i.e., the possibility that a program may be correct for one case of the conditional but not the other. To emphasize this situation, we say that the interval $a\boxed{\qquad b}$ is <u>regular</u> when $b - a \ge 0$, or <u>irregular</u> when $b - a < 0$. It is evident that a nonempty interval is always regular, but the empty interval can be either regular or irregular. (This is a slight abuse of language; it is really the interval diagram, rather than the interval itself, which is regular or irregular.)

From interval diagrams, we can build more complex entities called partition diagrams, which describe relationships between intervals. If $a_0, a_1, \ldots, a_n$ are expressions denoting integers, then:

(a) $a_0\boxed{\quad a_1\,\big|\,\cdots\; a_{n-1}\,\big|\quad a_n}$ is called a <u>partition diagram</u>.

(b) $a_0\boxed{\quad a_1}, \ldots, a_{n-1}\boxed{\quad a_n}$, i.e. the intervals denoted by diagrams obtained by eliminating all but an adjacent pair of lines, are called the <u>component intervals</u> of the partition diagram.

(c) $a_0\boxed{\quad a_n}$, i.e. the interval denoted by the diagram obtained by eliminating interior lines, is called the <u>total interval</u> of the partition diagram.

(d) The partition diagram is a logical expression which is true iff the component intervals are a partition of the total interval, i.e., iff the component intervals are disjoint and their union is the total interval.

As with interval diagrams, $\cdots\lfloor a \cdots$ may be written in place of $\cdots a{-}1\rfloor\cdots$, and $\cdots a\rfloor\cdots$ in place of $\cdots\lfloor a\quad a\rfloor\cdots$. Thus for example, $\boxed{a\quad b|\quad c}$ is a partition diagram which is true iff the component intervals $\boxed{a\quad}b = \{i \mid a \le i < b\}$, $\boxed{b} = \{b\}$, and $b\boxed{\quad c} = \{i \mid b < i \le c\}$ are disjoint and their union is the total interval $\boxed{a\quad c} = \{i \mid a \le i \le c\}$.

The nature of partitions implies that the size of the total interval is the sum of the sizes of the component intervals:

$$a_0\boxed{\quad a_1 \mid \cdots\ a_{n-1} \mid \quad a_n} \text{ implies } \#\, a_0\boxed{\quad a_n} = \sum_{i=1}^{n} \#\, a_{i-1}\boxed{\quad a_i} \, . \tag{2.2}$$

As shown in the Appendix, (2.2) implies the following fundamental property of partition diagrams:

$$a_0\boxed{\quad a_1 \mid \cdots\ a_{n-1} \mid \quad a_n} \text{ iff either} \tag{2.3}$$

$$a_0 \le a_1 \le \cdots \le a_{n-1} \le a_n \quad\text{or}\quad a_0 \ge a_1 \ge \cdots \ge a_{n-1} \ge a_n \, .$$

Note that the first inequality asserts that every component interval is regular, while the second inequality asserts that every component interval is empty.

From (2.3), the following simple cases are obvious:

$a\boxed{\quad b}$ is always true. (2.4)

$\boxed{a\quad b}$ iff $\boxed{a\quad b}$ iff $a \le b$ iff $\boxed{a\quad b}$ is nonempty. (2.5)

$\boxed{a\quad b|\quad c}$ iff $a \le b \le c$ iff $b \in \boxed{a\quad c}$. (2.6)

More interestingly, one can easily derive several "diagrammatically natural" rules of inference: (Here "line" refers to any vertical line in a diagram, including its associated expression.)

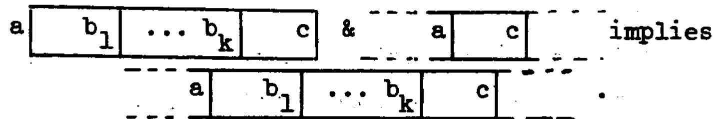
<u>Erasure</u> From a partition diagram one can infer any diagram obtained by deleting a line, i.e., (2.7)

$$\cdots a\rfloor\cdots \text{ implies } \cdots\cdots\cdots \, .$$

<u>Adjacent Duplication</u> From a partition diagram one can infer any diagram obtained by replicating a line next to itself, i.e., (2.8)

$$\cdots a\rfloor\cdots \text{ implies } \cdots a\rfloor\quad a\rfloor\cdots \, .$$

<u>Substitution</u> From two partition diagrams such that the end lines of the first match some pair of adjacent lines in the second, one can infer the diagram obtained by substituting the first diagram for the adjacent lines in the second: (2.9)



The use of these rules is illustrated by the following inferences, which will be pertinent to the binary search example given later:

(a) For any integers ℓ and u, (2.4) and (2.8) show that $\boxed{\ell \;\vert\; \ell \quad u \;\vert\; u}$ holds.

(b) Suppose $\boxed{\ell \;\vert\; a \quad b \;\vert\; u}$ and $a \le j \le b$. Then by (2.6) and (2.9), $\boxed{\ell \;\vert\; a \;\vert\; j \;\vert\; b \;\vert\; u}$ holds. In turn, by (2.7), this implies $\boxed{\ell \;\vert\; j \;\vert\; u}$, $\boxed{\ell \;\vert\; j{+}1 \quad b \;\vert\; u}$, and $\boxed{\ell \;\vert\; a \quad j{-}1 \;\vert\; u}$.

3. Functions as Array Values

There are two quite different concepts of an array. The more traditional view is that an array of, say, real numbers is a function from subscripts into variables, which in turn possess real values. The more recent view, expounded by Hoare(5,6) and Dijkstra(4), is that an array of real numbers is a variable whose value is a function from subscripts into real numbers. In this paper, we take the latter view. The effect is to banish the possibility of "sharing" or "aliasing" among array elements, which would greatly complicate the problems of proving program correctness.

Specifically, we assume that an array declared by τ **array** X(a:b) is a variable whose values range over the set of functions from the interval $\boxed{a \quad b}$ into the set τ.

For any function X, we write **dom** X for the domain of X and, when this domain is an interval, **lower** X and **upper** X for the integers such **dom** X = $\boxed{\textbf{lower}\ X \quad \textbf{upper}\ X}$. This definition of **lower** and **upper** is intentionally incomplete for the case where X is the unique function, denoted by <>, whose domain is the empty set.

When $S \subseteq$ dom X, we write $X \upharpoonright S$, called the restriction of X to S, to denote the function such that

$$\text{dom}(X \upharpoonright S) = S \qquad (3.1)$$

$$(\forall i \in S)\ (X \upharpoonright S)(i) = X(i)\ . \qquad (3.2)$$

This concept, which mirrors the informal idea of (the value of) a subarray or segment of an array, satisfies

$$\text{If } S' \subseteq S \subseteq \text{dom } X \text{ then } (X \upharpoonright S) \upharpoonright S' = X \upharpoonright S' \qquad (3.3)$$

$$X \upharpoonright \{\} = \langle\rangle\ . \qquad (3.4)$$

As an example, consider the program

```
begin integer i; integer array Squares(-5: 5);
integer array Possquares(0: 5);
integer array Nosquares(14: 5);
for i := -5 until 5 do Squares(i) := i × i;
for i := 0 until 5 do Possquares(i) := i × i;
...
end
```

At the program point indicated by the ellipsis, the following assertions will hold:

dom Squares = |-5 5|

lower Squares = -5

upper Squares = 5

(∀i ε |-5 5|) Squares(i) = i × i

Possquares = Squares ↾ |0 5|

Nosquares = Squares ↾ {} = <>

lower Nosquares > upper Nosquares .

The expressions lower X and upper X occur so frequently in interval and partition diagrams that it is useful to adopt conventions for eliding them unambiguously. When an interval or partition diagram is labelled with a function X, lower X may be omitted from the right of the leftmost line of the diagram, and upper X may be omitted from the left of the rightmost line. For example, X: | |a b| | stands for |lower X |a b| upper X|. Moreover, when an interval diagram is used to restrict a function X, the label can also be elided. For example, X ↾ | a| stands for X ↾ |lower X a|.

For a function X, we write {X}, called the <u>image</u> of X, to denote the set $\{X(i) \mid i \in \mathrm{dom}\ X\}$ of values obtained by applying X to members of its domain. (On the other hand, when x is not a function, {x} will denote the singleton set containing x.) Thus for example,

$$\{\underline{\mathrm{Possquares}}\} = \{0, 1, 4, 9, 16, 25\}$$
$$\{\underline{\mathrm{Possquares}} \rceil \boxed{1 \quad 3}\} = \{1, 4, 9\}$$
$$\{\underline{\mathrm{Squares}} \rceil \boxed{-2 \quad 2}\} = \{0, 1, 4\} \quad .$$

It is easily seen that images possess the following properties:

$$S \subseteq \mathrm{dom}\ X \text{ implies } \{X \rceil S\} \subseteq \{X\} \tag{3.5}$$
$$\{\langle\rangle\} = \{\} \tag{3.6}$$
$$S \cup S' = \mathrm{dom}\ X \text{ implies } \{X\} = \{X \rceil S\} \cup \{X \rceil S'\} \tag{3.7}$$
$$\{X \rceil \boxed{i}\} = \{X(i)\} \tag{3.8}$$
$$\# \{X\} \leq \# \mathrm{dom}\ X \ . \tag{3.9}$$

4. Operations on Relations

Suppose ρ is a binary relation between two sets U and U'. Then ρ^*, called the pointwise extension of ρ, is the binary relation between the set of subsets of U and the set of subsets of U', such that $S \rho^* S'$ holds if and only if $x \rho x'$ holds for all x in S and x' in S'.

When U and U' are both the set of integers, ρ could be any of the relational operators of Algol. For example, $\{2, 3\} \leq^* \{3, 4\}$ and $\{2, 3\} \neq^* \{4, 5\}$ are both true, while $\{2, 3\} <^* \{3, 4\}$, $\{2, 3\} =^* \{2, 3\}$, and $\{2, 3\} \neq^* \{2, 3\}$ are all false. The last two examples demonstrate that $\neq^*$ is not the negation of $=^*$ (and thereby show the importance of making * explicit).

The pointwise extension of any relation satisfies the following laws:

$$S \rho^* S' \ \ \& \ \ T \subseteq S \text{ implies } T \rho^* S' \tag{4.1a}$$
$$S \rho^* S' \ \ \& \ \ T' \subseteq S' \text{ implies } S \rho^* T' \tag{4.1b}$$
$$\{\} \rho^* S' \tag{4.2a}$$
$$S \rho^* \{\} \tag{4.2b}$$
$$(S \cup T) \rho^* S' \text{ iff } S \rho^* S' \ \ \& \ \ T \rho^* S' \tag{4.3a}$$
$$S \rho^* (S' \cup T') \text{ iff } S \rho^* S' \ \ \& \ \ S \rho^* T' \tag{4.3b}$$
$$\{x\} \rho^* \{x'\} \text{ iff } x \rho x' \ . \tag{4.4}$$

Occasionally, one needs the pointwise extension of a relation with regard to only a single argument. The simplest way of encompassing this case is to regard $x\ \rho^*\ S'$ as an abbreviation for $\{x\}\ \rho^*\ S'$ and $S\ \rho^*\ x'$ as an abbreviation for $S\ \rho^*\ \{x'\}$.

Another concept involving relations, somewhat more specialized than pointwise extension, is ordering. The usual idea of an ordered array can be generalized to an arbitrary relation in a way which unifies several important cases. Let X be a function whose domain is a set of integers, and let ρ be a binary relation appropriate to the type of result of X. Then X is ordered with regard to ρ, written $\mathbf{ord}_\rho\ X$, if and only if, for all i and j in the domain of X, $i < j$ implies $X(i)\ \rho\ X(j)$.

The following "orderings" appear as specific cases:

$\mathbf{ord}_\leq\ X$: increasing order

$\mathbf{ord}_<\ X$: strict increasing order

$\mathbf{ord}_\geq\ X$: decreasing order

$\mathbf{ord}_>\ X$: strict decreasing order

$\mathbf{ord}_=\ X$: all elements equal

$\mathbf{ord}_\neq\ X$: all elements distinct

Moreover, the generalization satisfies the following essential laws of ordering:

$$\mathbf{ord}_\rho\ X\ \&\ S \subseteq \mathbf{dom}\ X \text{ implies } \mathbf{ord}_\rho(X \rceil S) \tag{4.5}$$

$$\#\ \mathbf{dom}\ X \leq 1 \text{ implies } \mathbf{ord}_\rho\ X \tag{4.6}$$

$$\begin{aligned} &\text{If } S \cup T = \mathbf{dom}\ X\ \&\ S <^* T \text{ then} \\ &\quad (\mathbf{ord}_\rho\ X \text{ iff } (\mathbf{ord}_\rho(X \rceil S)\ \&\ \mathbf{ord}_\rho(X \rceil T)\ \&\ \{X \rceil S\}\ \rho^*\ \{X \rceil T\})) . \end{aligned} \tag{4.7}$$

An important special case of (4.7) is obtained by taking S and T to be two components of a partition:

$$\begin{aligned} &\text{If } X\colon \boxed{\quad k \mid \quad} \text{ then} \\ &\quad (\mathbf{ord}_\rho\ X \text{ iff } (\mathbf{ord}_\rho(X \rceil \boxed{\quad k}) \ \&\ \mathbf{ord}_\rho(X \rceil k\boxed{\quad}) \\ &\qquad \&\ \{X \rceil \boxed{\quad k}\}\ \rho^*\ \{X \rceil k\boxed{\quad}\})) . \end{aligned} \tag{4.8}$$

5. Binary Search

We have now introduced enough of our notation to demonstrate its use in describing - precisely yet intelligibly - why a program works. As an example, we describe an algorithm for binary search.

Given an ordered array X and a test value y, the program should set the boolean variable *found* to indicate whether any element of X is equal to y. If *found* is true, then the integer variable j should be set to the subscript of X such that X(j) = y. More precisely, if $\mathrm{ord}_{\leq}$ X, then executing the program should achieve the goal

if *found* **then** X: [___ |j| ___] & X(j) = y **else** {X} $\neq^{*}$ y .

Throughout program execution, *found* will only be set to true if X: [___ |j| ___] & X(j) = y is achieved. On the other hand, when *found* is false, it will not be known that y occurs nowhere in X, but only that it does not occur in either of two segments at the left and right ends of X. If we use the local variables a and b to delineate these segments, we have the invariant:

if *found* **then** X: [___ |j| ___] & X(j) = y
else X: [___ |a b| ___] & {X ↿ ([___]a ∪ b[___])} $\neq^{*}$ y .

On the one hand, this invariant can be achieved initially by setting *found* to **false** and making the end segments of X empty. On the other hand, the invariant implies the goal of the program if either *found* is true or a > b, since the latter condition implies that [a b] is empty and thus, from the partition diagram, [___]a ∪ b[___] = **dom** X. Thus our program has the form:

```
begin integer a, b;
a := lower X; b := upper X; found := false;
while ¬(found or a > b) do ...
end .
```

When execution of the body of the **while** statement begins, both the invariant and the **while** test will be true. Since a ≤ b, we can perform an operation "Pick j" (whose details will be considered later) which sets j to some integer in [a b]. At this stage, we will have

X: [___ |a ___ |j| ___ b| ___] & {X ↿ ([___]a ∪ b[___])} $\neq^{*}$ y ,

and we can compare X(j) with y. There are three cases:

(1) If X(j) = y, the invariant will be preserved if <u>found</u> is set to **true**.

(2) If X(j) < y, then $\text{ord}_{\leq}$ X insures that $\{X \rceil \boxed{\quad j}\} <^* y$, so that $\{X \rceil (\boxed{\quad j} \cup b\boxed{\quad})\} \neq^* y$. This permits us to set a to j + 1.

(3) If X(j) > y, then a similar argument justifies setting b to j - 1.

Thus our program is:

```
begin integer a, b;
a := lower X; b := upper X; found := false;
while ¬(found or a > b) do
    begin
    "Pick j";
    if X(j) = y then found := true else
        if X(j) < y then a := j + 1 else b := j - 1
    end
end
```

Termination is guaranteed by the fact that each iteration either sets <u>found</u> to **true**, which immediately stops further iterations, or else decreases the size of $\boxed{a \quad b}$, whose emptiness will cause termination. The absence of subscript errors is guaranteed since X: $\boxed{\quad \boxed{j} \quad}$ holds at the program points where X(j) is evaluated.

It should be noticed that this description of binary search does not exclude the possibility that $\boxed{\text{lower X} \quad \text{upper X}}$, and therefore $\boxed{a \quad b}$, might be irregular. The heart of the matter is the reasoning about partition diagrams, which was formalized at the end of Section 2. One of the virtues of this kind of reasoning is that it includes the irregular case without any special case analysis.

To complete our program, we must digress from the topic of arrays to specify "Pick j". In this case, the problem is not to find a correct realization - either j := a or j := b would be correct - but to find an efficient one. The need to shrink $\boxed{a \quad b}$ as much as possible suggests choosing j at or near the midpoint of $\boxed{a \quad b}$, i.e., j := (a + b) ÷ 2.

However, we must be sure that, if $a \leq b$, then $j := (a + b) \div 2$ will achieve $a \leq j \leq b$, despite the fact that integer division involves rounding (and that the details of this rounding might vary for different machines, especially when $a + b$ is negative). Fortunately, it is enough to know that division by two is a monotonic function which is exact for even numbers. For $a \leq b$ implies $a + a \leq a + b \leq b + b$, so that monotonicity gives $(a + a) \div 2 \leq (a + b) \div 2 \leq (b + b) \div 2$, and exactness for even numbers gives $a \leq (a + b) \div 2 \leq b$. (S. Winograd has pointed out that $j := (a + b) \div 2$ is unnecessarily prone to overflow, in comparison with, for example, $j := a + (b - a) \div 2$. We leave it to the reader to show that the correctness of his improvement can still be proved with a monotonicity argument.)

6. Array Assignment

We must now move beyond programs such as binary search which merely use arrays, to consider programs which change arrays. In programming languages at the level of Algol, the fundamental agent of change is an assignment statement which alters a single array element, e.g., $X(i) := e$.

Hoare[5,6] has shown that, to deal with this statement from the viewpoint that an array is a function-valued variable, we must regard it as an abbreviation for the assignment $X := [X \mid i \mid e]$, where $[X \mid i \mid e]$ denotes the function which is similar to X except that it maps i into e. More formally, $[X \mid i \mid e]$ is defined when $i \in \underset{\sim}{\text{dom}}\, X$, in which case it is the function satisfying

$$\underset{\sim}{\text{dom}}\, [X \mid i \mid e] = \underset{\sim}{\text{dom}}\, X \tag{6.1}$$

$$[X \mid i \mid e](i) = e \tag{6.2}$$

$$[X \mid i \mid e](j) = X(j) \quad \text{when } j \neq i\ , \tag{6.3}$$

and, as an immediate consequence of (6.3),

$$[X \mid i \mid e] \upharpoonright S = X \upharpoonright S \quad \text{when} \quad S \subseteq \text{dom}\, X \text{ and } i \notin S\ . \tag{6.4}$$

Once $X(i) := e$ is seen as an abbreviation for $X := [X \mid i \mid e]$, the usual axiom of assignment[2]:

$$P|_{x \to e}\ \{x := e\}\ P \tag{6.5}$$

(where $P|_{x \to e}$ denotes the result of substituting e for x in P) extends to an axiom of array assignment[6]:

$$P|_{X \to [X \mid i \mid e]} \{X(i) := e\} \; P \quad . \qquad (6.6)$$

(Because of (6.1), when this axiom is used, the substitution $X \to [X \mid i \mid e]$ need not be applied to occurrences of X in dom X, X:, lower X, or upper X.)

7. Equivalence Relations for Arrays

For many programs which alter arrays, such as sorting programs, a full specification will stipulate both that the final value of the array will possess some property, such as being ordered, and that the final value will be related to the initial value in some way, such as being a rearrangement. Often - even when the situation is intuitively obvious - a formidable technical apparatus is needed to formulate and prove the latter kind of specification.

To deal with these problems it is useful to introduce several equivalence relations for array values. Suppose X and Y are both functions whose domains are sets of integers. Then:

(a) We write $X \approx\!\!\!\sim Y$, and say that X is a redistribution of Y iff $\{X\} = \{Y\}$.

(b) We write $X \sim Y$, and say that X is a rearrangement of Y iff there is a bijection B (sometimes called a one-to-one correspondence or a permutation) from dom X to dom Y such that $(\forall i \in \text{dom } X)\; Y(B(i)) = X(i)$.

(c) We write $X \approx Y$, and say that X is a shift of Y iff there is a bijection as in (b) with the special form $B(i) = i + s$ for some integer s.

This defines an increasingly stringent sequence of equivalence relations. Thus, where ρ is $\approx\!\!\!\sim$, $\sim$, or $\approx$:

Transitivity $X \rho Y$ & $Y \rho Z$ implies $X \rho Z$ (7.1)

Symmetry $X \rho Y$ implies $Y \rho X$ (7.2)

Reflexivity $X \rho X$ (7.3)

$X \approx Y$ implies $X \sim Y$ (7.4)

$X \sim Y$ implies $X \approx\!\!\!\sim Y$. (7.5)

Finally, we have three more specific laws. Exchanging a pair of elements produces a rearrangement:

$$(\forall i,\ j \in \underset{\sim}{\mathrm{dom}}\ X)\,[[X \mid i \mid X(j)] \mid j \mid X(i)] \sim X\ , \tag{7.6}$$

two one-element arrays with equal values are shifts of one another:

$$\boxed{i} = \underset{\sim}{\mathrm{dom}}\ X\ \&\ \boxed{j} = \underset{\sim}{\mathrm{dom}}\ Y\ \&\ X(i) = Y(j) \text{ implies } X \simeq Y\ , \tag{7.7}$$

and a shift of an ordered array is ordered:

$$X \simeq Y\ \&\ \underset{\sim}{\mathrm{ord}}_\rho\ X \text{ implies } \underset{\sim}{\mathrm{ord}}_\rho\ Y\ . \tag{7.8}$$

As Hoare has pointed out,[3] for any program which only alters an array by performing exchanges, (7.1), (7.3), and (7.6) are sufficient to show that the final array value is a rearrangement of the initial value. However, to deal with programs which move information from one array to another, we must also consider the concatenation of array values.

8. Concatenation

Let X and Y be functions whose domains are intervals with sizes m and n respectively. Then $X \frown Y$, called the <u>concatenation</u> of X and Y is the unique function such that

$$\underset{\sim}{\mathrm{dom}}\ (X \frown Y) = \boxed{1 \quad m{+}n}$$

$$(X \frown Y) \rceil \boxed{1 \quad m} \simeq X$$

$$(X \frown Y) \rceil \boxed{m{+}1 \quad m{+}n} \simeq Y\ .$$

The choice of one as a lower bound is arbitrary, since we will always regard shifts of concatenated array values as equivalent.

Let <> denote the unique function whose domain is empty. Then concatenation satisfies the following laws:

$$X \frown \langle\rangle \simeq X \tag{8.1}$$

$$\langle\rangle \frown X \simeq X \tag{8.2}$$

$$(X \frown Y) \frown Z \simeq X \frown (Y \frown Z) \tag{8.3}$$

$$X \simeq X'\ \&\ Y \simeq Y' \text{ implies } X \frown Y \simeq X' \frown Y' \tag{8.4}$$

$$X \frown Y \sim Y \frown X \tag{8.5}$$

$$X \sim X'\ \&\ Y \sim Y' \text{ implies } X \frown Y \sim X' \frown Y' \tag{8.6}$$

$$X: \boxed{\;\;a\;\;\boxed{\;\;}} \text{ implies } X \simeq (X \upharpoonright \boxed{\;\;a}) \frown (X \upharpoonright a\boxed{\;\;}) \qquad (8.7)$$

$$\{X \frown Y\} = \{X\} \cup \{Y\} \qquad (8.8)$$

$$\underset{\sim}{\text{ord}}_\rho(X \frown Y) \text{ iff } \underset{\sim}{\text{ord}}_\rho X \;\&\; \underset{\sim}{\text{ord}}_\rho Y \;\&\; \{X\}\, \rho^* \,\{Y\} \;. \qquad (8.9)$$

The first four laws show that array values form a monoid under concatenation, provided that shift equivalence is used in place of true equality. The next two laws show that this monoid becomes commutative when the less stringent equivalence of rearrangement is used. (Technically, one can make these statements precise by working with the quotient of the set of array values under the equivalence relations $\simeq$ or $\sim$.)

The last three laws establish the basic connections between concatenation and partitions, images, and ordering. In particular, (8.9) is a consequence of (4.8) and (7.8).

In fact (8.3) actually remains true when $\simeq$ is changed to $=$. But the stronger relationship is irrelevant, since we should never be interested in true equality for concatenated array values.

9. Merging

As a second example of program description, we consider the problem of merging: Given two ordered arrays X and Y, set Z to an ordered rearrangement of the concatenation of X and Y. We assume that Z is just the right size to hold the result. Thus if

$$\underset{\sim}{\text{ord}}_\leq X \;\&\; \underset{\sim}{\text{ord}}_\leq Y \;\&\; \#\,\underset{\sim}{\text{dom}}\, Z = \#\,\underset{\sim}{\text{dom}}\, X + \#\,\underset{\sim}{\text{dom}}\, Y \;,$$

then executing the program should achieve the goal

$$\underset{\sim}{\text{ord}}_\leq Z \;\&\; Z \sim X \frown Y \;.$$

During execution, each array will be partitioned into a processed part on the left and an unprocessed part on the right, the processed part of Z will be an ordered rearrangement of the concatenation of the processed parts of X and Y, the unprocessed part of Z will be the right size to hold the unprocessed parts of X and Y, and all processed elements in Z will be smaller or equal to all unprocessed elements in X or Y. (The last condition is needed to insure that the unprocessed elements can be moved into Z without rearranging the already processed elements.) Thus we have the invariant:

$$I \equiv X: \boxed{\quad}\boxed{kx\quad} \;\&\; Y: \boxed{\quad}\boxed{ky\quad} \;\&\; Z: \boxed{\quad}\boxed{kz\quad} \tag{a}$$

$$\&\; \mathrm{ord}_{\leq}\, Z \upharpoonright \boxed{\quad}\,kz \tag{b}$$

$$\&\; Z \upharpoonright \boxed{\quad}\,kz \sim X \upharpoonright \boxed{\quad}\,kx \frown Y \upharpoonright \boxed{\quad}\,ky \tag{c}$$

$$\&\; \# Z: \boxed{kz\quad} = \# X: \boxed{kx\quad} + \# Y: \boxed{ky\quad} \tag{d}$$

$$\&\; \{Z \upharpoonright \boxed{\quad}\,kz\} \leq^{*} \{X \upharpoonright \boxed{kx\quad}\} \cup \{Y \upharpoonright \boxed{ky\quad}\}\;. \tag{e}$$

This invariant can be achieved initially by making the processed parts all empty, and it will imply the goal of the program when the unprocessed parts are all empty, which - by (d) - will occur when the unprocessed part of Z is empty. Thus we can use a program of the form:

```
begin integer kx, ky, kz;
kx := lower X; ky := lower Y; kz := lower Z;
while kz ≤ upper Z do "Copy One Element"
end .
```

In "Copy One Element", a single element will be moved from the unprocessed part of X or Y into the processed part of Z. To preserve condition (e) the element to be moved must be the smallest member of $\{X \upharpoonright \boxed{kx\quad}\} \cup \{Y \upharpoonright \boxed{ky\quad}\}$. Since both X and Y are ordered, this will be the smaller of the leftmost unprocessed elements, X(kx) or Y(ky), providing both unprocessed parts are nonempty. However, if only one unprocessed part is nonempty, its leftmost element will be the element to be moved. Thus:

```
"Copy One Element" ≡
    if (if kx > upper X then false else
        if ky > upper Y then true else
        X(kx) ≤ Y(ky))
    then "Copy X" else "Copy Y"   ,
```

where, prior to executing "Copy X",

$$IX \equiv Z: \boxed{kz}\boxed{\quad} \;\&\; X: \boxed{kx}\boxed{\quad} \tag{f}$$

$$\&\; X(kx) \leq^{*} \{X \upharpoonright \boxed{kx\quad}\} \cup \{Y \upharpoonright \boxed{ky\quad}\} \tag{g}$$

will hold as well as the invariant I.

Thus (e) will be preserved if "Copy X" moves X(kx) out of the unprocessed part of X and into the processed part of Z. Moreover, (e) insures that X(kx) will be larger or equal to the elements which have previously been moved into Z. Thus the ordering (b) will be preserved if X(kx) is placed at the right of the processed part of Z. Therefore:

"Copy X" ≡

```
begin Z(kz) := X(kx); kx := kx + 1; kz := kz + 1 end ,
```

and by a similar argument

"Copy Y" ≡

```
begin Z(kz) := Y(ky); ky := ky + 1; kz := kz + 1 end .
```

Formally, in the notation of Reference 2, "Copy X" must meet the specification

I & IX {"Copy X"} I .

To exemplify the application of the various laws we have stated, we give a formal proof of this specification. The assignment axioms (6.5) and (6.6) imply I' {"Copy X"} I, where

I' ≡ $I|_{kz \to kz+1}\ |_{kx \to kx+1}\ |_{Z \to [Z\ |\ kz\ |\ X(kx)]}$

= X: [kx|] & Y: [|ky] & Z: [kz|] (a')

& $ord_{\leq}$ [Z | kz | X(kx)] ↿ [kz] (b')

& [Z | kz | X(kx)] ↿ [kz] ∼ X ↿ [kx] ⌒ Y ↿ [ ]ky (c')

& # Z: kz[ ] = # X: kx[ ] + # Y: [ky] (d')

& {[Z | kz | X(kx)] ↿ [kz]} $\leq^*$ {X ↿ kx[ ]} ∪ {Y ↿ [ky]} (e')

(Here we have written kx| instead of |kx+1 and kz| instead of |kz+1.) Thus we must show that I & IX implies I', i.e., that lines (a) through (g) imply (a') through (e').

By the rule (2.9) of substitution, (a) and (f) imply

X: [|kx|] & Y: [|ky] & Z: [|kz|] (h)

which, by the rule (2.7) of erasure, implies (a') as well as various partition diagrams used in the sequel. In particular, by (2.2) and (2.1), X: [kx|] implies # X: [kx] = # X: kx[ ] + 1, and Z: [kz|] implies # Z: [kz] = # Z: kz[ ] + 1, so that (d) implies (d').

Next, we have

$[Z \mid kz \mid X(kx)] \upharpoonright \boxed{\quad kz}$

$\approx [Z \mid kz \mid X(kx)] \upharpoonright \boxed{\quad}kz \frown [Z \mid kz \mid X(kx)] \upharpoonright \boxed{kz}$ $\qquad$ $(Z\text{: } \boxed{\quad}\!\boxed{kz})$, (8.7)

$= Z \upharpoonright \boxed{\quad}kz \frown [Z \mid kz \mid X(kx)] \upharpoonright \boxed{kz}$ $\qquad$ (6.4)

$\approx Z \upharpoonright \boxed{\quad}kz \frown X \upharpoonright \boxed{kx}$ $\qquad$ (7.7),(8.4),(6.2)

$\sim (X \upharpoonright \boxed{\quad}kx \frown Y \upharpoonright \boxed{\quad}ky) \frown X \upharpoonright \boxed{kx}$ $\qquad$ (c),(8.6)

$\sim (X \upharpoonright \boxed{\quad}kx \frown X \upharpoonright \boxed{kx}) \frown Y \upharpoonright \boxed{\quad}ky$ $\qquad$ (8.3),(8.5),(8.6),(7.4)

$\approx X \upharpoonright \boxed{\quad kx} \frown Y \upharpoonright \boxed{\quad}ky$ $\qquad$ $(X\text{: } \boxed{\quad}\!\boxed{kx})$, (8.7)

which establishes (c'), and also

$$[Z \mid kz \mid X(kx)] \upharpoonright \boxed{\quad kz} \approx Z \upharpoonright \boxed{\quad}kz \frown X \upharpoonright \boxed{kx} \ . \qquad \text{(i)}$$

Then

$\{[Z \mid kz \mid X(kx)] \upharpoonright \boxed{\quad kz}\}$

$= \{Z \upharpoonright \boxed{\quad}kz \frown X \upharpoonright \boxed{kx}\}$ $\qquad$ (i),(7.4),(7.5)

$= \{Z \upharpoonright \boxed{\quad}kz\} \cup \{X \upharpoonright \boxed{kx}\}$ $\qquad$ (8.8)

$= \{Z \upharpoonright \boxed{\quad}kz\} \cup \{X(kx)\}$ $\qquad$ (3.8)

$\leq^* \{X \upharpoonright \boxed{kx\quad}\} \cup \{Y \upharpoonright \boxed{ky\quad}\}$ $\qquad$ (e),(g),(4.3a)

$= \{X \upharpoonright \boxed{kx} \frown X \upharpoonright kx\boxed{\quad}\} \cup \{Y \upharpoonright \boxed{ky\quad}\}$ $\qquad$ $(X\text{: } \boxed{kx}\!\boxed{\quad})$, (8.7)

$= \{X \upharpoonright \boxed{kx}\} \cup \{X \upharpoonright kx\boxed{\quad}\} \cup \{Y \upharpoonright \boxed{ky\quad}\}$ $\qquad$ (8.8)

so that (4.1a) and (4.1b) give (e') and

$$\{Z \upharpoonright \boxed{\quad}kz\} \leq^* \{X \upharpoonright \boxed{kx}\} \ . \qquad \text{(j)}$$

Finally, (3.1), (2.1), and (4.6) imply $\text{ord}_{\leq}\, X \upharpoonright \boxed{kx}$, which with (b), (j), and (8.9) implies $\text{ord}_{\leq}(Z \upharpoonright \boxed{\quad}kz \frown X \upharpoonright \boxed{kx})$, which with (i) and (7.8) implies (b').

10. Multidimensional Arrays

Although the concepts we have presented were developed and tested in the context of one-dimensional arrays, most of them extend to the multidimensional case. The major additional concept which is needed is the *Cartesian product*:

$$S_1 \times \dots \times S_n \equiv \{\langle i_1, \dots, i_n \rangle \mid i \in S_1 \; \& \dots \& \; i_n \in S_n\} .$$

A Cartesian product of intervals is called a *block*. The values of the array declared by τ **array** $X(a_1 : b_1, \dots, a_n : b_n)$ are functions whose domain is the block $\boxed{a_1 \quad b_1} \times \dots \times \boxed{a_n \quad b_n}$.

It is evident that the values of subarrays of X such as rows and columns are restrictions of X to certain blocks. For example, the following asserts that $\langle i, j \rangle$ is a saddle point of the two-dimensional array X:

$$\{X \rceil (\boxed{i} \times \boxed{})\} \le^* X(i, j)$$
$$\& \; X(i, j) \le^* \{X \rceil (\boxed{} \times \boxed{j})\} .$$

11. Conclusion

The contents of this paper is only a small beginning. It is largely limited to one-dimensional integer-subscripted arrays, and even within this domain further study is certain to produce significant extensions and changes. But we have gone far enough to demonstrate the value of the underlying approach: We have formulated concepts, laws, and notations which are powerful tools for the precise yet intelligible description of a significant aspect of programming.

Hopefully, this work suggests guidelines for further progress: One should focus upon particular mechanisms such as arrays rather than generalities which pertain to all computation. Concepts and laws are more fundamental than notation per se, and should reflect intuitive understanding. Most important, the crucial test is the ability to describe real programs in a way which is not only precise but also intelligible to the human reader.

APPENDIX

Proof of Proposition (2.3)

We leave it to the reader to verify that either $a_0 \leq a_1 \leq \cdots \leq a_n$ or $a_0 \geq a_1 \geq \cdots \geq a_n$ implies $a_0\boxed{ a_1}\boxed{\cdots\ a_n}$. The following proof of the converse was found by F. L. Morris.

Suppose $a_0\boxed{ a_1}\boxed{\cdots\ a_n}$. From (2.2) we have

$$\# a_0\boxed{ a_n} = \sum_{i=1}^{n} \# a_{i-1}\boxed{ a_i} , \qquad \text{(a)}$$

where

$$\# a\boxed{ b} = \underline{\text{if}}\ b - a \geq 0\ \underline{\text{then}}\ b - a\ \underline{\text{else}}\ 0$$

is always nonnegative and is zero iff $a\boxed{ b}$ is empty. For arbitrary a_i's simple cancellation gives

$$a_n - a_0 = \sum_{i=1}^{n} a_i - a_{i-1} .$$

Then subtraction of (a) from both sides gives

$$f(a_0, a_n) = \sum_{i=1}^{n} f(a_{i-1}, a_i) , \qquad \text{(b)}$$

where

$$f(a, b) = b - a - \# a\boxed{ b} = \underline{\text{if}}\ b - a \geq 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ b - a .$$

is always nonpositive and is zero iff $a\boxed{ b}$ is regular.

The interval $a_0\boxed{ a_n}$ must be either empty or regular (or both). Suppose it is empty. Then (a) asserts that a sum of nonnegative terms is zero, which implies that each term is zero. Thus for each i, $a_{i-1}\boxed{ a_i}$ is empty, and $a_{i-1} \geq a_i$.

On the other hand, suppose $a_0\boxed{ a_n}$ is regular. Then (b) asserts that a sum of nonpositive terms is zero, which implies that each term is zero. Thus for each i, $a_{i-1}\boxed{ a_i}$ is regular, and $a_{i-1} \leq a_i$.

ACKNOWLEDGEMENTS

I am indebted to the members of IFIP Working Group 2.3, who have provided motivation, inspiration, and helpful criticism. I am also grateful for the hospitality of the University of Edinburgh and the support of the Science Research Council during the period when this paper was written.

REFERENCES

1. Floyd, R. W. "Assigning Meanings to Programs," Proceedings of Symposia in Applied Mathematics 19, American Mathematical Society, Providence (1967), pp. 19-32.

2. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming," Comm. ACM 12, no. 10, October 1969, pp. 576-581.

3. Hoare, C. A. R. "Proof of a Program: FIND," Comm. ACM 14, no. 1, January 1971, pp. 39-45.

4. Dijkstra, E. W. A Discipline of Programming, Prentice-Hall, 1976.

5. Hoare, C. A. R. "Notes on Data Structuring," in Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press (1972), pp. 83-174.

6. Hoare, C. A. R., and Wirth, N. "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica 2, 1973, pp. 335-355.

7. Hoare, C. A. R., "A Note on the FOR Statement," BIT 12, no. 3, 1972, pp. 334-341.

8. Cooper, D. C. "Proofs about Programs with One-Dimensional Arrays," unpublished.

9. Burstall, R. M. "Some Techniques for Proving Correctness of Programs which Alter Data Structures," Machine Intelligence 7, November 1972, pp. 23-49.