NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Collection-Oriented Languages

Jay M. Sipelstein   Guy E. Blelloch

September 17, 1990
CMU-CS-90-127 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Several programming languages arising from widely varying practical and theoretical considerations share a common high-level feature: their basic data type is an aggregate of other data types and their functional primitives operate on these aggregates. Examples of such languages and the collections they support are FORTRAN 90 and arrays, APL and arrays, Connection Machine LISP and xectors, PARALATION LISP and paralations, and SETL and sets. Acting on large collections of data with a single operation is the hallmark of data-parallel programming and massively parallel computers. These languages—which we call *collection-oriented*—are thus ideal for use with massively parallel machines, even though many of them were developed before parallelism and associated considerations became important. This paper examines collections and the operations that can be performed on them in a language-independent manner. It also critically reviews and compares a variety of collection-oriented languages with respect to their treatment of collections, gives many examples and code fragments from these languages, and elucidates certain problems that may arise when defining and implementing collection operations.

This research was supported in part by the Defense Advanced Research Projects Agency (DOD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20 and in part by the National Science Foundation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the National Science Foundation or the U.S. government.

510.7808
C28r
90-1217
c.2

Keywords: massively parallel programming, data parallelism, collection-oriented language

1 Introduction

We call a programming language *collection-oriented* if aggregate data structures and operations for manipulating them are primitives in the language. Sets, arrays, lists, and vectors are common collections supported by these languages. Typical collection-oriented operations include the componentwise multiplication of two vectors, summing the elements of a sequence of numbers, and transposing a matrix. Many standard languages—such as C, PASCAL, AND FORTRAN 77—allow the user to operate on large sets of data with explicit loops. However, in a collection-oriented language, collections and operations on them are directly supported by the language. Table 1 shows how typical collection operations are specified in several collection-oriented languages.

Operations on collections are in most cases ideally suited for implementation on massively parallel machines. Elements of collections can be distributed across the available processors—in the limit, each processor holds only a single value. Then, for example, to perform the componentwise multiplication of two vectors, corresponding elements are assigned to the same processor, and in standard Single-Instruction Multiple-Data (SIMD) fashion, each processor multiplies corresponding values. An operation taking linear serial time thus may be easily converted to a constant time parallel operation. Summing the elements of a sequence quickly in parallel is also straight-forward. Each processor is assigned an element. Then in logarithmic time, the elements are summed with a tree-like computation: at each step half the elements remaining are removed by pairing up adjacent values and adding them. An operation like transposing a matrix can be accomplished using the communication facilities of the parallel machine: each processor computes the location to which its piece of the data must go and then sends it there.

The exact mechanism used to carry out each of these procedures will be machine specific, but the user of a collection-oriented language need not know these low-level architectural details. Only the language implementor need worry about how to efficiently map the collection operations onto a particular parallel machine. High-level collection operations therefore give the programmer easy access to massive parallelism.

In view of this, it is no surprise that almost all high-level languages developed for massively parallel machines are collection-oriented. The available high-level languages for the Connection Machine [9, 24]—C* [18], *LISP [15], Connection Machine LISP (or CM-LISP) [21], and PARALATION LISP [19]—all are collection-oriented. Likewise Parallel PASCAL [5] for the MPP [2] is collection-oriented. More recently the vector extensions to FORTRAN 90 were designed with modern parallel and vector computers in mind. Variants of FORTRAN 90 [28] have been implemented for the CM, the MPP and DAP 500 [17]. FORTRAN 90 is also collection-oriented.

It might be surprising however, that collection-oriented languages have been developed independently of and prior to parallel machines. The programming language community has long recognized that aggregate data structures and general operations on them give great flexibility to programmers and implementors of a language. This idea is the basis (or one of the bases) for a great many programming languages which were designed long before parallel machines. The first such language, APL [12], appeared in the early 60's and utilized a compact notation for representing array operations.[1] In the 70's, several other collection-

[1] APL and APL2 are used as generic terms in this paper. The former refers to any version of APL without nested collections and the later refers to any of the newer dialects including APL2 [11] and Dictionary APL [13].

UNIVERSITY LIBRARIES
CARNEGIE MELLON UNIVERSITY
PITTSBURGH, PA 15213-3890

FP	

$(/+)\circ(\alpha\times)\circ$`trans`	`A = [1 0 5 3]`
	`B = [3 4 3 7]`
	`⇒ 3 + 0 + 15 + 21 = 39`

Compute the dot product of two vectors

APL	

$+/(A*(*\backslash 1,(((\rho A)-1)\rho x)))$	`A = [1 2 3 4]`
	`x = 2`
	$\Rightarrow 1+2\times 2+3\times 2^2+4\times 2^3 = 41$

Evaluate a polynomial with given coefficients A at value x

CM-LISP	

`(let ((l (length A))`
`(p (α (β+ A →1.0) αl)))`
`(- (β+ (α* p (αlg p)))))`
	`A = [a b c a d c b d]`
	`⇒ 2`

Compute Shannon entropy of A: $H(i) = -\sum p(i)\lg p(i)$
where $p(i)$ is the probability that i occurs in the input string, for each i.

SETL	

`a := [2..n];`
`result := {};`
`loop while #a > 0 do`
`p := first a;`
`a := [x in a | (x mod p) /= 0];`
`result := result with p;`
`end;`
`print result;`
	`N = 10`
	`⇒ [2 3 5 7]`

Find prime numbers with the Sieve of Erastosthenes

FORTRAN 90	
`R[2:n-1] =` `(F[1:n-2]-2*F[2:n-1]+F[3:n])/(d*d)`	`F = [1 2 2 3 4]` `R[2:4] ⇒ [-.1 .1 0]`

Compute the second derivative of F given a vector of values

Table 1: Examples of operations on collections in a variety of languages

oriented languages were developed, including SETL [20], FP [1], and NIAL [14]. Each has features that are in large part derived from APL. Many modern functional languages such as HASKELL [10] and MIRANDA [26] also have collection-oriented features—in particular set and list comprehensions. Now, in the late 80's and early 90's, with the advent of massively parallel machines, many researchers have been working on compiling these older languages for these new machines [4, 8, 27, 7].

Our goals for this paper are twofold. First, we want to acquaint the reader with the various sorts of collections and with the wide variety of collection operations supported by existing languages. Primitive operations examined in detail include applying a function to all the elements of a collection (apply-to-each), using a function to combine the elements of a collection together (reduce), extracting elements satisfying certain properties from a collection (select or set comprehension) and rearranging the elements of a collection (permute). We hope to impress upon the reader the utility of these operations for solving many computational problems.

Second, we wish to introduce language designers to the spectrum of collection-oriented languages and to some of the design decisions and problems that may occur in attempts to specify such languages. The expressibility of a particular collection-oriented language is determined by two factors: the collections the language supplies and the applicability and generality of the associated collection operations. A variety of languages (including APL2, SETL, CM-LISP, PARALATION LISP, FORTRAN 90) are critically compared in this regard. Code fragments from each language demonstrate how/whether these languages support specific collection operations. We bring up interesting language specific issues as they arise.

The paper is organized as follows. Section 2 gives some extended examples of collection operations in several languages in order to provide the reader with a taste of the problems this paper explores. Section 3 introduces a taxonomy of collections. Section 4 examines in great detail a wide variety of collection operations. Finally Section 5, explores the differing treatment of collections in specific collection-oriented languages.

2 Example Collection-oriented Operations

This section illustrates four collection-oriented operations in several different languages. The purpose of these examples is both to give an overview of various collection-oriented languages and to introduce many of the important issues discussed in detail in the paper. These four examples are shown in Table 2 and are implemented in APL, SETL, PARALATION LISP, CM-LISP and FORTRAN 90. Each code fragment is quite concise in comparison to the equivalent code in a conventional language. These examples are described at a high enough level that parallel implementation is possible if parallel descriptions of the collection operations are available.

Unary Apply-to-each

An operator that applies a function to each element of a collection is an *apply-to-each*. The first example in Table 2 shows how the negate function can be applied over the elements of a collection. These languages specify apply-to-each in a variety of ways. In APL and FORTRAN 90, just placing the negate symbol in front of a vector signifies that each element should be negated. This syntax is called an *implicit apply-to-each*, since there is no explicit declaration that the negate should be applied over the elements. In CM-LISP it is necessary to place an α in front of the negate; the α can be thought of as taking the function and distributing it over

Language	Code
APL:	`-A`
CM-LISP:	`(α- A)`
SETL:	`{-e : e in A}`
PARALATION LISP:	`(elwise ((e A))` `  (- e))`
FORTRAN 90:	`-A`

```
A  =  [4    5   -2   11   -7]
   ⇒  [-4  -5    2  -11    7]
```

Example 1: Negating each element of a collection

Language	Code
APL:	`A + B + 2`
CM-LISP:	`(α+ (α+ A B) α2)`
SETL:	`[A[i] + B[i] + 2` `  : i in domain(A)]`

PARALATION LISP:	`(elwise ((e1 A)`
	`         (e2 B))`
	`  (+ (+ e1 e2) 2))`

FORTRAN 90	`A + B + 2`

```
A  =  [4   5   2  11   7]
B  =  [2   5   1   4   6]
   ⇒  [8  12   5  17  15]
```

Example 2: Summing corresponding elements of two collections, and a scalar

Language	Code
APL:	`A[P]`
CM-LISP:	`(β@ P A)`
SETL:	`[A[i] : i in P]`
PARALATION LISP:	`(<-A :by` `  (match P (index A)))`
FORTRAN 90.:	`A[P]`

```
A          =  [r  e  s  e  t]
P          =  [1  2  4  3  0]
CM-LISP    ⇒  [t  r  e  e  s]
Others     ⇒  [e  s  t  e  r]
```

Example 3: Permuting a collection; P is a permutation of the indices of A

Language	Code
APL2:	`+/¨A`
CM-LISP:	`(αβ+ A)`
SETL:	`[+/e : e in A]`
PARALATION LISP:	`(elwise ((e A))` `  (vref :with '+ e))`

```
A  =  [[3   4   2]  [2  8]]
   ⇒  [9   10]
```

Example 4: Summing subcollections of a nested collection

Table 2: Some basic collection-oriented operations

the collection. This notation was taken from FP. We call this form an *explicit apply-to-each*. In SETL and PARALATION LISP, it is necessary to bind a variable name to a representative element of a collection and then apply the negate to this variable. One can think of the expression as stating: "for each e in A, negate e." We call this form, a *binding apply-to-each*. In SETL, the form is actually a special case of the more general set/tuple comprehension primitive discussed in Section 4.2.3. What effect do these three forms have on a collection-oriented language? We argue, for example, that implicit apply-to-each interacts badly with overloading of functions based on argument type (see Section 4.2.1).

Non-unary Apply-to-each

The second example in Table 2 demonstrates the case of applying the function + (addition) over the corresponding elements of two collections and then adding the constant 2 to each element of the results. Unlike the first example, the function takes more than a single argument, and one of the arguments is not a collection; it is a scalar. From this example, two new issues arise: *element correspondence* and *argument extension*.

What does the phrase "corresponding elements of two collections" in the previous paragraph mean? Intuitively, we can think of lining up the two collections and applying the function + at each location. But what if the two collections cannot be "lined up" (they may be different lengths)? Or, what if the collections are not ordered (as with sets in SETL)? All the languages considered in Table 2 have a different way of defining apply-to-each on multiple argument functions. APL requires the two arguments be of equal length. SETL requires the use of an explicit index set. PARALATION LISP requires the two arguments come from the same paralation; this is an even stronger requirement than being of the same length. CM-LISP puts no requirements on the relationship between the two arguments—it adds elements which have the same key.

Another issue raised by this example is how to define what it means to "add a scalar to a collection"? There must be some mechanism for specifying that the scalar should be treated as a collection, each of whose elements has that particular value. We call this *argument extension*. In CM-LISP this is accomplished via the same α form used for specifying an apply-to-each; this is called *explicit extension*. In APL scalars are automatically extended as needed; this is called *implicit extension*. Implicit argument extension can lead to ambiguity when the collections to be extended are nested (see Section 4.2.2).

Rearranging Elements

In addition to having some way of applying a function to each element of a collection, collection-oriented languages supply operations that can rearrange the elements of a collection. The third example in Table 2 illustrates a *permute* operation. The permute operation rearranges the elements of a collection according to a collection of indices. Permute is an example of an operation that has different definitions in different languages: APL performs a permute that is the mathematical inverse of that performed by CM-LISP. Permute is a special case of indexing elements in APL, FORTRAN 90 and SETL and the permutation indices all refer to the result collection. In CM-LISP these indices refer to the argument collection. In languages like CM-LISP with more complex collection types, the permute can be generalized greatly (Section 4.1.2).

Nested Collections and Operators

The final example demonstrates the utility of nested collections. A is a collection of collections, and the sum of the elements in each of these subcollections is needed. Not all collection-oriented languages allow nested collections: FORTRAN 90 and APL both do not permit them, although APL2 does. For computing with nested collections, those languages supporting them include explicit extensions to avoid possible ambiguities. Example 4 combines the apply-to-each concept with the operation of summing the elements of a collection. The latter computation is described in all of these languages as a plus *reduction*. Reduction (Section 4.1.1) is one example of a high-order function: it takes as arguments both a combining function and a collection to be reduced. Each of the collection-oriented languages in these examples, except FORTRAN 90, possess such operators.

3 Collections

The exact definition of a collection varies greatly from language to language. The simplest and most general characterization of a collection is that it is a group of objects viewed as a whole [16]. This captures the intent of our usage: collection-oriented programming languages should be able to encapsulate a group of objects into a collection and then manipulate this conglomeration of elements in useful ways. It is the methods of encapsulation and manipulation that are interesting. This section surveys the different kinds of collections that collection-oriented languages support.

3.1 General Classification of Collections

We categorize collections along three axes: what kinds of elements are allowed, whether or not these elements may be of mixed type, and whether the elements are implicitly and/or explicitly ordered within the collection.

Elements of Collection

The greatest distinction between the kinds of collections that a language supports is that between *simple* collections and *nested* collections. The elements of a simple collection may not be collections themselves. Languages that support only simple collections include APL and FORTRAN 90. Nested collections are the most general type of collection: they may have collections as elements. Nested collections are useful for representing data more complex than vectors or sets such as trees and segmented vectors [3]. Languages permitting nested collections include APL2, SETL, CM-LISP, and PARALATION LISP.

A useful subclass of the simple collections are the *structure* collections. An element is a structure if it has a fixed number of fields and the only operations that can be performed on the element are extraction and insertion of a field (for example, a PASCAL record, or a C or LISP structure). Both CM-LISP and PARALATION LISP support structure collections.

Type Homogeneity of Elements

An issue orthogonal to the types of the elements allowed in a collection is the question of whether and how elements of differing types may be combined in the same collection. A

Language	Relation Among Elements Types	
	heterogeneous	*homogeneous*
CM-LISP	✓	
SETL	✓	
PARALATION LISP	✓	
APL		✓
APL2	✓	
FORTRAN 90		✓
NIAL	✓	

Language	Types of Element		
	atomic	*structure*	*collection*
CM-LISP	✓	✓	✓
SETL	✓		✓
PARALATION LISP	✓	✓	✓
APL	✓		
APL2	✓		✓
FORTRAN 90	✓		
NIAL	✓		✓

Language	Ordering			
	unordered	*linear-ordered*	*grid-ordered*	*key-ordered*
CM-LISP	xet	xector	—	xapping
SETL	set	tuple	—	map
PARALATION LISP	—	field	—	—
APL	—	—	array	—
APL2	—	—	array	—
FORTRAN 90	—	—	array	—
NIAL	—	—	array	—

Table 3: The collection types available in various collection-oriented languages. A sequence-ordered collection is a vector. In CM-LISP, *xets* and *xectors* are considered special cases of *xappings*. In SETL, *maps* are considered special cases of *sets*: a *map* is a *set* of two element *tuples*.

homogeneous collection is one in which all elements have the same type; a *heterogeneous* collection has no such constraint.

The characterization of a collection as being heterogeneous or homogeneous can only be made relative to the type system of the underlying language. For example, consider the following nested collection of integers:

[[5 12 13] [7 24 25] [12345 76199512 76199513] [1 2 3 4]].

In some languages the length of a collection might be part of the type of the collection. PASCAL has no type consistent with each element of the example above; `array [1..3] of integer` is fine for the first three elements, but not for the fourth. In such a language this collection is heterogeneous: the last subcollection is of a different length from the others and hence of different type. The type systems of most collection-oriented languages are not this stringent and to a lenient language this collection is just a vector of vectors and therefore homogeneous. To push this example further, if 16 bit and 32 bit integers were of distinct types in a language (say `integer` and `long integer`), then the third subcollection might itself be considered heterogeneous.[2]

Table 3 shows the type homogeneity of some collection-oriented languages.

Collection Ordering

Another important property of collections is whether or not there is an ordering associated with the positions of elements in the collection: can we say that one element comes before another in the collection? This is independent of the value of the element, and depends only on position within the collection. For example, the elements of an array in C are ordered (by their index), while the elements of a mathematical set are not. The nature of the ordering of a collection has a strong influence on the collection operations that can be defined on it (see Section 4.1.2).

We distinguish between four classes of collection orderings:

Unordered Unordered collections are essentially sets of elements, except that sets do not allow repetitions and a general unordered collection does.[3]

Sequence-Ordered Sequence-ordered collections are vectors. These are also called linearly-ordered collections.

Grid-Ordered Grid-ordered collections are arrays of arbitrary dimension. A 1-1 correspondence exists between ordered tuples of integers in some interval and elements of the collection.

[2]These kind of considerations can allow a compiler to use knowledge of the homogeneity of a collection to generate more efficient code. It can recognize homogeneous collections and maintain type knowledge of the entire collection. In particular, if a language only admits homogeneous collections, all collections can be given a succinct type name (of length at most the depth of nesting of the collection). The type information can be calculated once for the entire structure. In a strongly typed language, the alternative is to infer the type of each element of the collection as we need it; the type of the collection may then be as complex as the data structure itself. For example, if we want to perform an element by element add with two collections of elements of type `short integer`, it might be advantageous to call a separate routine that adds collections of shorts together, rather than call a general routine that adds two collections and has to infer the type of each element as it procedes. One implementation of PARALATION LISP by the second author takes advantage of this idea for a sizable efficiency gain.

[3]The distinction here is one of sets *vs.* multisets.

Key-Ordered Key-ordered collections are indexed via an arbitrary mapping function that has keys as its domain and values as its range. Some language further require all the keys of a collection to be unique.

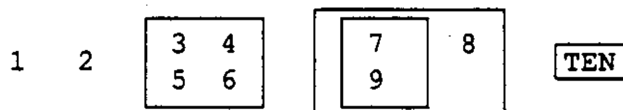
Unordered sets are the foremost collection type in SETL. Sequence-ordered collections are the basic data structure of LISP-like languages. Grid-ordered collections are the basic data structure of APL-like languages. Key-ordered collections are the most general since the domain of the mapping function can be a sequence of integers (giving a sequence-ordering) or can be tuples from a sequence of integers (for a grid-ordering). Table 3 shows the orderings supported by the languages under consideration and the names each language gives to these collections.

An additional kind of collection found in some functional languages is the infinite collection. In languages supporting either lazy or normal order evaluation it is possible to create collections that are potentially infinite in extent. Implementations of this collection generally only compute actual elements of the collection as they are needed by the functions acting upon them. As long as the manipulations deal with the collection itself, and not the individual elements, the implementation is free to avoid computing those elements. Languages supporting this feature include MIRANDA [26], HASKELL, and SCHEME [23]. Although the infinite collections in these particular languages are linearly ordered, this does not have to be true in general. For instance, it would be relatively easy in any of these languages to create the SETL-style set of the natural numbers: the collection would pick a new "random" natural number each time it is accessed, and would guarantee no repetitions. Similarly, it would be easy to build infinite collections of any of the other forms discussed in this section.[4]

3.2 Language Specific Collections

Table 3 summarizes the differences that exist between the collections supported by some collection-oriented languages. This section explores the distinctions between individual languages in greater detail.

The heterogeneous nestable array is the fundamental collection in APL2. This contrasts with APL, which only allows homogeneous simple arrays. The introduction of nested collections into APL2 allows multiple arguments and multiple results from user defined functions, as well as the combination of numeric and character data in vectors. APL2 also adds a great many new operators to APL for handling nested collections. The following APL2 collection contains vectors, strings and arrays (the boxes indicate nesting):



The fourth element is itself an array containing a scalar and a vector and the fifth is a vector of characters.

SETL supports three kinds of collections: the *set*, the *tuple* and the *map*. Sets are unordered and enclosed by { } (curly braces). Sets are constrained not to have duplicates and SETL enforces this constraint. The sets

```
{1,1,2}, {2,2,1,1,1}, {1,2}, {2,1,2,2}
```

[4]One nice example of this is FAC [25], a lazy functional version of APL with infinite, ragged arrays.

are all equivalent. Tuples are ordered and enclosed by [ ] (square braces). Maps are key-ordered collections represented as a set of ordered pairs (two element tuples); there may be multiple elements with the same key. Any valid SETL object can be an element of a set, a tuple, or the key or value field of a map, thus allowing arbitrary levels of nesting in collections.

The primitive collection of CM-LISP is the *xapping*. A xapping is an key-ordered heterogeneous collection. Each element is an ordered pair of the form

index→value.

`index` (also called the *key*) and `value` can be any LISP object (including another xapping) but the indices in a given xapping must all be distinct. One example of a xapping is

```
{ b→boat c→car a→apple }.
```

CM-LISP also provides syntactic sugar for vectors:

```
[one two three] ≡ { 1→one 2→two 3→three }.
```

This abbreviated form is called a *xector*. A second shorthand notation is available to describe sets:

```
{ one two three } ≡ { one→one two→two three→three }
```

This representation of a set is called a *xet*.

PARALATION LISP adds a new data type to COMMON LISP [22], the *paralation*, a contraction of "parallel" and "relation". A paralation consists of two parts: a fixed number of *sites* which are numbered from 0 and a dynamic number of *fields*. Each field of a paralation has a value for each site of the paralation. It is helpful to think of a paralation as a database and a field as holding a piece of data for every element of the database. A typical paralation with two fields looks something like:

index-field	name-field	year-field
0	King George III	1760
1	Washington	1789
2	Adams	1797
3	Jefferson	1801
⋮	⋮	⋮
40	Reagan	1981
41	Bush	1989

Fields are named (in this case, `name-field` and `year-field`) and field values may be heterogeneous and can be fields themselves—allowing nested collection. Paralations can be created in two ways. The `make-paralation` function creates a new paralation of given length and one field, the numbers from 0 to (length - 1). Alternatively, a field of a new paralation can be created using COMMON LISP reader syntax:

```
(make-paralation 5) ≡ #F(0 1 2 3 4).
```

Additional fields of an existing paralation can be created with the `elwise` function, which takes a list of fields in the same paralation, performs an elementwise computation on the

values of those fields, and returns a new paralation containing the results. This construct is discussed in depth in Section 4.2. An alternative way of thinking about paralations is to view them as types: many fields may be created in one paralation, but each must have the same length. Also, only fields of the same paralation may be included in the same `elwise`, even if their paralations have the same length. Paralations (or their fields) are essentially sequence-ordered, but PARALATION LISP also supports key-ordered operations on and between paralations.

FORTRAN 90 has extensions to FORTRAN 77 to provide support for arrays and vectors as fundamental data-structures. There is a sophisticated syntax for assigning elements into arrays and for indexing elements from arrays discussed in Section 5.3. The vector [1 2 3 4] can be denoted by both

```
(/ 1 2 3 4 /)
(/ I, I=1,4 /).
```

4 Collection Operations

A collection-oriented language is characterized by two traits: the kinds of collections supported by the language and the operations allowed for those collections. The previous section focused on the first of these issues; we now turn to the second. This section studies some collection-oriented operations and explores their uses. Different languages describe these operations in slightly different ways. Section 5 discusses specific languages in detail.

There are two classes of collection operations: *aggregate* operations and *apply-to-each* forms. The aggregate operations of a language are those operations that operate on a collection as a whole. These operations take collections as arguments, compute some function of the collections and return a scalar or new collection (Section 4.1). The apply-to-each construct acts as an iterator: given a collection as one argument and a function as another argument, it returns the result of the function being applied to each the elements of the collection separately (Section 4.2). This distinction can be quite fuzzy; what appears to be an apply-to-each in one language may be an aggregate operation in another. We emphasize that the space of collection-oriented operations has no simple topology. The classification scheme used in this paper is only one of several possible.

The collection operations described here should be thought of as abstract mathematical constructs. The result of performing a collection operation should depend only on the semantics of that operation in the language and not on the language's implementation. This statement may seem obvious, but it has non-obvious consequences. For example, whether or not a particular program is run on a parallel or serial machine, or whether a particular algorithm is used in the implementation of an operation should not affect the results of the calculation. Parallel implementation of an apply-to-each would be disallowed or have unpredictable results if the apply-to-each causes multiple side-effects to the same location. Problems such as this have resulted in the historic difficulty of defining language semantics for parallel machines.

4.1 Aggregate Operations

The aggregate operations are those operations that explicitly compute on a collection as a whole; this computation cannot be broken down into elementwise application of other

functions. There are many types of collection operations and they are categorized here according to the kinds of collections on which they operate.

4.1.1 Generic Operations

Generic operations are the most general aggregate operations. There are few restrictions concerning the kinds of collections to which they can be applied. All operations discussed in this section apply equally well to both ordered and unordered collections.

Perhaps the simplest generic operations are those returning basic information about a collection. One familiar example of this is the length operation that returns the length (number of components) of a collection:[5]

```
A  ⟹ [3  1  4  1  5  9]
(length A) ⟹ 6.
```

This very useful operation is something most non-collection languages do not support. For example, there is no way to find the length of an array in C given a pointer to the array. Such information must either be passed as an extra parameter to functions needing it or be specified as a separate field in the data structure. Other examples of this class of information operations might include predicates and functions that describe the exact kind of collection being used, for example, (set? A), and (nested? A).

A useful generic aggregate operation is select. select takes collection and predicate arguments and creates a collection containing those elements of a collection satisfying the predicate:

```
(select even?  [1  2  3  4  5]) ⟹ [2  4]
(select prime?  [1  2  3  4  5]) ⟹ [2  3  5].
```

select is closely related to the pack operation described in the next section and can be implemented in terms of it if the collection argument is ordered. We defer further discussion until then.

A powerful generic operation is *restricted reduction*. The reduction operator (reduce) takes as arguments a collection C and a binary function f that is associative and commutative. reduce returns the result of combining the elements of C with f:[6]

```
(reduce f [a  b  c  d])≡ (f a (f b (f c d)))
(reduce + [2  4  6  8]) ⟹ 20
(reduce max [2  4  6  8]) ⟹ 8.
```

The requirement that f be associative and commutative guarantees that the result of a restricted reduction be the same, regardless of the manner by which it might be evaluated. This means a compiler is free to convert the first example to

```
(f (f a b) (f c d)), or (f b (f (f c a) d))
```

[5]Disclaimer: We will be using LISP notation throughout this paper in our non-language-specific examples. This is merely a syntactic convenience to avoid ambiguity and is not meant as an endorsement.

[6]reduce gets its name from its use in APL. In APL reduce applied to an array combines elements along the final dimension of the array, *reducing* the overall dimension of the array by one.

or any other grouping or ordering. This allows possible parallel implementation using a tree-like evaluation. Typical uses of reduce are: summing the elements of a collection, finding the minimum or maximum element of a collection, and using logical reduction to determine the and or or of a boolean collection. Some languages may impose restrictions on the functions f over which reductions may be performed: APL and FORTRAN 90 only allows a fixed set of reductions, while APL2, CM-LISP, PARALATION LISP, and SETL allow reduction on any binary function. See Section 5 to see the way different languages support reduction on generic collections.

4.1.2 Ordered vs Unordered Collections

An important distinction between ordered and unordered collections is that the former makes possible an unambiguous correspondence between elements of two differing collections. If two collections are ordered in the same manner and are of equal sizes, we may "superimpose" one collection on top of another by merging elements with the same index or key. Such collections are said to be of the same *shape* or to be *conformable*. In this section, we assume both collection arguments to a binary collection operation are conformable. In Section 4.2, we consider what happens when these conditions are not met and discuss generalizations of "conformable".

set operations

One common kind of collection is the set. Most of the collection-oriented languages examined here support the standard corpus of set operations on collections: intersection, union and member?.

(union [1 2 3] [2 3 4]) ⟹ [1 2 3 4] or [1 2 3 2 3 4]
(intersection [1 2 3] [2 3 4]) ⟹ [2 3]
(member? 2 [1 2 3]) ⟹ True.

Although languages like SETL enforce a no-repetition constraint on sets, the exact definition of a set is orthogonal to the support of set operations in a language. These set operations can be defined analogously on multisets (as in the union example).

join **and** pack

A natural operation on two collections is to join one collection with another. If these collections are ordered, join might simply concatenate one collection to the end of the other. If they are not ordered, join could act like a set union (with or without the removal of any duplicate entries). For example, any of the following are reasonable outcomes of a join operation:

M	⟹	[m i c k e y]
(join M [m o u s e])	⟹	[m i c k e y m o u s e]
	or	[c e e i k m m o s u y]
	or	[m i c k e y o u s].

The actual output is determined by the particular language used and by the specific ordering imposed on M.

Closely related to the select operation defined in the previous section is the pack function (also called compression in the APL community). This function's arguments are two ordered

collections of the same shape: one containing data and another containing boolean values. The result of the pack is another collection consisting of those elements of the first collection for which the corresponding element of the second collection is `true` (or `1`):

(pack [m i c k e y] [1 1 1 0 1 0]) ⟹ [m i c e].

pack can be used to implement select by applying a predicate to all the elements of some collection to generate the booleans and then pulling out the elements satisfying the predicate. An important difference between these operations is that pack depends on the ordering of the collection arguments while select does not.

permute

Thus far, none of the operations considered rearrange the order of the elements in a collection. This is accomplished with the permute function. The arguments to permute are two ordered collections of the same shape: the data collection and the index collection. The latter is a collection whose elements are some permutation[7] of the indices for the first collection. The result of the operation is a collection in which the index collection specifies where the corresponding element of the data collection goes in the result collection. Expressed in C syntax,

```
out[P[i]] == in[i],
```

where `P` is the permutation vector and `in` and `out` are the input and output vectors respectively. For example:[8]

(permute [l e a s t s] [6 5 2 3 1 4]) ⟹ [t a s s e l].

The inv-permute operator works the same way as permute, except that the index vector specifies where the corresponding result element comes from, instead of where it goes:

```
out[i] == in[P[i]]
```

(inv-permute [l e a s t s] [6 5 2 3 1 4]) ⟹ [s t e a l s].

inv-permute can be generalized to the case where the index collection is not a proper permutation of the indices of the data collection: each element can be put into its proper position as long as the elements of the index collection are a subset of the indices of the data collection:

(inv-permute [m i c k e y] [2 1 1 2 4 5]) ⟹ [i m m i k e]
(inv-permute [m i c k e y] [1 5]) ⟹ [m e]
(inv-permute [m i c k e y] [1 2 3 4 5 6 1]) ⟹ [m i c k e y m].

In a language with key-ordered collections, the permute and inv-permute operators can be defined analogously. The index collection of a permute is a collection of items whose keys are the same as the keys of the data and whose values are all distinct. The result is a collection whose keys are the values of the index collection and whose values are the values of the data collection and the pairing up is done according to the keys of both collections:

[7] No index is repeated and all are present.

[8] We use the convention that indices start at 1.

(permute {1→a 2→b 3→c} {1→2 2→3 3→1}) ⟹ {1→c 2→a 3→b}.

This is exactly what we would expect if we viewed sequence-ordered collections as key-ordered collections with their indices as keys. In this formalism, inv-permute just switches the roles of the key and value of the index collection: the result has keys from the *keys* of the index collection, and has values from the values of the data collection and the pairing up is according to the *values* of the *index* collection (compare with above):

(inv-permute {1→a 2→b 3→c} {1→2 2→3 3→1}) ⟹ {1→b 2→c 3→a}.

This definition of a key-ordered permute can be extended to cover the cases when the sets of keys for the index and data collections are not the same by simply omitting the keys not in both:

(permute {a→x b→y c→z} {a→5 b→6 d→7}) ⟹ { 5→x 6→y }.

A further generalization can be made to the case where the values of the index collection are not distinct.[9] permute can take an extra argument specifying a way to resolve collisions (elements that are supposed to be moved to locations with the same key). In CM-LISP this argument is a function and colliding elements are combined with this function:

(permute {1→17 2→19 3→23} {1→1 2→5 3→1} +) ⟹ {1→40 5→19}

This key-ordered, collision-resolving permute is a powerful operation and generalizes many of the operations given so far. For example:

(reduce A f) ≡ (permute A { 1→1 2→1 ⋯ n→1 } f)

where reduce is the restricted version of reduction discussed earlier (Section 4.1.1).

reduce and scan

Section 4.1.1 examined the restricted reduce operator for unordered collections which required the combining function to be commutative and associative. When operating on an linearly-ordered collection, the combining function need not be commutative; the result of the reduce is defined to be that obtained when the values are combined in the same order as they appear in the collection. The two expressions

(reduce f [1 2 3 4]) ⟹ (f (f (f 1 2) 3) 4)
(reduce f [1 2 3 4]) ⟹ (f (f 1 2) (f 3 4)).

are valid reductions and will return the same result because of the associativety of f, but the expression

(f (f 4 3) (f 2 1))

is not valid, since f may not be commutative.

[9]This is for languages where the index set of a key-ordered collection must have distinct values.

The scan operator is a generalization of reduce that is only defined on linearly ordered collections.[10] scan returns the prefix computation of an ordered collection C and a binary associative function f: that is, it returns a collection whose ith element is the reduce of the first i elements of C by f. For example,

(scan + [3 1 4 1 5 9]) ⟹ [3 4 8 9 14 23].

Note that the value of (reduce f A) is the last element of the result.

Another issue to consider is that the underlying language may have its own rules governing associativity. Suppose we define reduce on linearly ordered collections so that the combining function is "put in text":

(reduce f [1 2 3 4]) ⟹ 1 f 2 f 3 f 4.

The evaluation of this expression depends on whether the language or function in question is right or left associative. Indeed, since APL is right associative and SETL is left associative,

(reduce - [1 2 3 4]) ⟹ -2 APL
⟹ -8 SETL.

Consider the following operation:

(reduce + [bignum -bignum smallnum -smallnum]),

where + is floating point addition and bignum and smallnum are numbers close to the maximum and minimum representable positive floating point numbers. Floating point addition is not associative. Hence, if the collection argument is unordered, the result of evaluating this expression will depend on how the additions get parenthesized: there is no *correct* answer. Even if the collection *is* ordered, the additions could still be grouped in differing manners, with correspondingly different results. The above "in text" rule can be used to define an unambiguous result for reduce and scan of linearly ordered collections with non-associative functions by uniformly declaring an evaluation order. This, of course, suffers from the same problem with parallel implementation discussed previously.

4.2 Apply-to-each

The *apply-to-each* forms are the second major class of collection operations. Apply-to-each forms act as iterators by calling for the application of a function to every element of a collection. The result of an apply-to-each operation is a collection whose shape is the same as the argument collection. This kind of operation maps perfectly to the massively parallel programming paradigm. A simple example of apply-to-each is negating each element of a collection (see Table 2, example 1).

There are two styles of apply-to-each in collection-oriented languages: *extension* and *binding*. In the binding apply-to-each[11] a generic element of a collection is given a name and the computation to be performed on that element is described. This is the method used by PARALATION LISP (the elwise statement) and by SETL (in the guise of set comprehension).

[10] This is not strictly true. One can define versions of scan for trees in analogous ways. [3]

[11] There does not appear to be a standard term for this in the literature.

On the other hand, extensions modify the evaluation of a function so that it operates over the elements of a collection. These extensions are specified in one of two ways: *implicitly* or *explicitly*. In some languages (APL, for example), extensions are performed automatically if the operation in question needs them in order to be well defined: this is the implicit case. Alternatively, explicit extension requires some mechanism to describe precisely those functions and/or arguments that must be extended. The tradeoff between these alternatives is largely one of convenience and conciseness *vs* lack of ambiguity. Extensions are used by APL, FORTRAN 90, CM-LISP, and FP.

Probably the most important difference between collection-oriented languages is whether user-defined functions are permitted as the functional argument to an apply-to-each form. All the languages under consideration in this paper with explicit apply-to-each allow any function, whether primitive or user-defined, to be so used. The languages with implicit apply-to-each restrict the functions to a fixed set of primitive operators. It is no coincidence that these languages, FORTRAN 90 and APL, have very primitive type schemes, no polymorphism, and no nested collections; they are able to get away with implicit apply-to-eachs and yet incur no ambiguity.

4.2.1 Function Extension and Unary Functions

First we see what complications develop in the fairly straightforward case of apply-to-each with functions taking a single argument. Consider a simple example: suppose we have a collection A consisting of five integers. What should the value of (square A) be, where square of an integer returns its square? One possible way of defining this result is to create a collection identical to A and then replace each element with its square:

```
A ⟹ [1 2 3 4 5]
(square A) ⟹ [1 4 9 16 25].
```

This can be viewed as extending the domain of the function square from integers to collections of integers—or alternatively, of overloading the function definition for collection types. To compute the value of the function applied to a collection, we apply it to each element separately and wrap the results back into a collection, preserving any ordering. In the same manner, the domain of negate could be extended:

```
B ⟹ [2 3 5 7 11]
(negate B) ⟹ [-2 -3 -5 -7 -11].
```

In general, suppose f is a unary function of type $O_1 \rightarrow O_2$, where O_1 and O_2 are some set of objects. The domain of f can then be extended to C_{O_1}, the collections containing elements of O_1, by:

$$f([x_1 \quad x_2 \quad \cdots \quad x_n]) \equiv [f(x_1) \quad f(x_2) \quad \cdots \quad f(x_n)].$$

The resulting collection has the same shape as the argument collection; if the input is unordered, so is the output.

These examples use implicit functional extension: no extra notation is needed to get the apply-to-each of the function. Some languages require an explicit apply-to-each construct to indicate that an apply-to-each operation should be performed. In FP and CM-LISP explicit apply-to-each is used and both these languages denote apply-to-each operations with the

symbol α. This paper borrows their notation. With explicit extension, the square example becomes:

(αsquare A) ⟹ [1 4 9 16 25].

Although it leads to increased code size, explicit functional extension is needed to remove possible ambiguity. Consider the square example again and suppose that it occurred in the context of some linear algebra code. In languages that allows operator overloading, it would be reasonable to add a definition of square for vectors that calculates the inner product of an argument with itself. In this case our example evaluates:

(square A) ⟹ 55.

Explicit functional extension allows whichever behavior is desired to be achieved. The previous example is still valid with this overloading; the α defines which version of square to use.

Particular care must be taken with nested collections in order to avoid ambiguity. Consider a nested collection C of three collections, each of which is a collection of three elements:

C ⟹ [[1 2 3] [4 5 6] [7 8 9]].

What should the value of (reverse C) be? There are at least two possible solutions, depending on the level of nesting at which reverse acts: we may apply reverse to the whole collection, or we may go into each element and reverse it. These two cases may be disambiguated with an explicit apply-to-each:

(reverse C) ⟹ [[7 8 9] [4 5 6] [1 2 3]]
(αreverse C) ⟹ [[3 2 1] [6 5 4] [9 8 7]].

The second example should be read as "apply reverse to the elements of C."

A binding apply-to-each form can also be used to describe an apply-to-each operation. This is an explicit construct detailing the computation to be performed on each element of the collection. It is similar in form to a loop over the elements of a collection, but there are no explicit loop bounds. In PARALATION LISP, the binding apply-to-each is denoted with the key-word elwise, which we adopt for this paper. elwise takes a list of pairs and a function body as arguments. Each pair consists of the name of a collection and a dummy name for a representative element of the collection. All collections in a single elwise must be conformable. The function body uses the dummy names as variables. Using this notation, the square example given previously becomes:

```
(elwise ((a A)) (square a))
```

which should be read as "take each element a of A and square it." The second instance of the nested reverse example is written:

```
(elwise ((a A)) (reverse a)).
```

The set comprehension primitive of SETL and HASKELL is another way to denote a binding apply-to-each. In SETL, the reverse example is expressed by:

```
[reverse a : a in A],
```

which is read as "create a tuple consisting of the reverse of each element a in A."

From a purely notational perspective, both kinds of explicit apply-to-eachs, extension and binding, have advantages and disadvantages over their implicit counterpart. The primary advantage is one of clarity: code is quite clean and easy to read, and there is no ambiguity about the operations being performed. Unfortunately, for trivial operations, or when there is no possibility of ambiguity, inserting the extra syntax becomes tedious.

From the perspective of the compilation process, explicit apply-to-eachs are superior to implicit ones. Explicit apply-to-eachs exactly specify the depth of nesting at which a function should be applied. If a language is not strongly typed, it may be very difficult to do the necessary type inference to implicitly extend functions at compile time. This is one of the major difficulties in creating APL compilers: in APL there is no general mechanism for deciding the dimension of an array before runtime so it is impossible for a compiler to generate the correct code for a function call in all cases without making the call so general that it becomes inefficient. It may be necessary to generate code for the various cases and test at runtime [6]. With explicit extension, a compiler can decide how the apply-to-each should be computed, up to possible polymorphism/overloading of function names, even if the exact type of the collection and its elements are not known.

A second issue that we have been avoiding until now is side-effects. The result of performing an apply-to-each with a side-effecting function is problematical. Nowhere have we mentioned an order of evaluation, either implied or actual, for the apply-to-each form; in fact, we have discussed the utility of apply-to-each for specifying data parallelism. To get around this problem a language may explicitly define an ordering for the evaluation of an apply-to-each (SETL does this with tuple formers), or it may explicitly say that the result of such an operation is undefined (PARALATION LISP), or it may impose restrictions to the functional arguments (APL).

4.2.2 Argument extension and non-unary functions

The preceding section discussed issues that arise when we only consider unary functions. How can we extend these ideas to n-ary functions? In this case, the primary new issues are *argument extension*.[12] and comformability. Given an operation like `(+ A B)`, where A and B are collections of integers, what must be the relationship between A and B? What if A is an integer and B is a collection? What if A and B are unordered or nested?

For now we limit ourselves to the binary case, in which the collections are nested lists or vectors. One possible way of defining apply-to-each for binary functions is to proceed in the same manner as for unary functions. This means extending the definition of the binary function so that it takes a collection of ordered pairs as arguments and combines each pair with the function:

$$f([(x_1,\ \ y_1)\ \ (x_2, y_2)\ \ \cdots\ \ (x_n,\ \ y_n)]) \equiv [f(x_1, y_1)\ \ f(x_2, y_2)\ \ \cdots\ \ f(x_n, y_n)]$$

This definition is used by FP and generalizes very nicely to n-ary functions; instead of collections of pairs we can have collections of n-tuples. Unfortunately this does not *really* solve the binary case. What we have done is change the definition of f from a binary function to

[12]Once again, there seems to be no standard terminology for this.

a unary function whose single argument is a pair, and then apply the new unary function to the collection.

An alternative approach is to have f group corresponding elements (elements having the same key or index) of the argument together and create a new collection with the same structure as the original:

$$f([x_1 \quad x_2 \cdots \quad x_n],[y_1 \quad y_2 \cdots \quad y_n]) \equiv [f(x_1,y_1) \quad f(x_2,y_2) \quad \cdots \quad f(x_n,y_n)]$$

So for example:

```
A        ⟹ [1  2  3  4  5]
B        ⟹ [2  3  5  7  11]
(+ A B)  ⟹ [3  5  8  11  16]
(* A B)  ⟹ [2  6  15  28  55].
```

This element-by-element generalization of a binary function, and the obvious extension to n-ary functions, is the method used in all the collection-oriented languages, except FP, under consideration in this paper.

Once again collection-oriented languages may use explicit function extension, instead of the implicit extension just shown, to indicate this kind of operation. In this case, the same examples are written:

```
(α+ A B)  ⟹ [3  5  8  11  16]
(α* A B)  ⟹ [2  6  15  28  55].
```

Examining the definition of binary apply-to-each more closely reveals a few tacit assumptions. First note that the collections under consideration must be ordered. There is an inherent matching-up of indices that cannot occur if such indices are not present. This really is not *too* undesirable when it is remembered that the primary unordered collection is the set: trying to perform an elementwise addition on the elements of two sets does seem like a primitive set operation (in fact, SETL overloads + to union if its arguments are sets). The issue of performing apply-to-each operations on sets will be considered at greater length later in this section.

Another issue to consider is what happens when the collections being operated upon do not have the same structure. This breaks up into two different cases. The first is when the index/key sets of the collections are not identical; *e.g.* when the collections are of different lengths. One way to handle this situation is to simply signal a runtime error. Both APL and FORTRAN 90 do this. Languages that do define this operation try to make it meaningful. In particular, on intersection of the index sets, the apply-to-each should have the standard functionality. Thus a new collection should be created whose index set is the intersection of the index sets of the arguments. The values of the answers should be correct for these indices. For vectors of different lengths, this means making a new vector whose length is the minimum of the argument vector lengths and whose elements represent the apply-to-each of the truncated vectors.

```
(+ [2  3] [4  5  6  7]) ⟹ [6  8].
```

The problem is what to do with the rest of the elements. There are a few possibilities. The extra elements can just be dropped (CM-LISP does this). Alternatively, the new index set

could be the union of the index sets of the arguments, and any elements with the same index are combined into the result:

```
(+ [2 3] [4 5 6 7]) ⟹ [6 8 6 7].
```

Both these solutions have the problem that they are fairly arbitrary: there is no real reason to prefer one to the other. One way to finesse this problem is to prevent it from ever occurring. This situation never arises in PARALATION LISP because `elwise` can only be used for fields of the same paralation, which are guaranteed to be conformable.

The other case is when the collection arguments to an apply-to-each are a collection and a single value (or scalar). The collection-oriented languages under consideration here all interpret this to mean combine all the elements of the collection with this value:

$$(\texttt{f s } [\texttt{x}_1\ \texttt{x}_2\ \cdots\ \texttt{x}_n]) \Longrightarrow [(\texttt{f s x}_1)\ \ (\texttt{f s x}_2)\ \cdots\ (\texttt{f s x}_n)].$$

This allows the following:

```
B        ⟹ [2 3 5 7 11]
(+ B 2)  ⟹ [4 5 7 9 13]
(* B 7)  ⟹ [14 21 35 49 77].
```

Another way of looking at this is to say that we *argument extend* the scalar `s` by converting it into a collection all of whose elements are `s` and whose shape is that of the other argument. This is implicit argument extension. As with function extension, there is an explicit version as well:

```
(α* B α7).
```

The α may be thought of as specifying that enough copies of the function and scalars are created to match the shape of the collection argument, at which point each function is applied to its arguments. The CM-LISP manual [21] discusses this idea in depth.

It may seem that if a language allows explicit function extension then there is no need for any explicit argument extension. In particular, in the previous example the α7 seems unnecessary. Since `*` has been extended, the interpreter or compiler can deduce that `7` must be extended as well. However, just as with implicit function extension, implicit argument extension may sometimes result in ambiguity that may require explicit extension for clarification. An example using nested collections demonstrates this:

```
               A  ⟹  [[2 3] [4 5]]
               B  ⟹  [[6 7] [8 9]]
    (append A B)  ⟹  [[2 3] [4 5] [6 7] [8 9]]
   (αappend A B)  ⟹  [[2 3 6 7] [4 5 8 9]]
  (αappend A αB)  ⟹  [[2 3 [6 7] [8 9]] [4 5 [6 7] [8 9]]]
  (αappend αA B)  ⟹  [[[2 3] [4 5] 6 7] [[2 3] [4 5] 8 9]].
```

Each of these results has a very different collection structure. In general without some sort of explicit argument extension, it may be impossible to specify which of these results is desired.

Ambiguity can also result if overloaded operators are present in the language. A nested version of the square from the beginning of section 4.2.1 is an example of this. If square is overloaded to compute inner products when given a vector argument, all the problems of the

append case crop up, plus the confusion with regard to which square is actually being applied (the vector version or the scalar version).

4.2.3 Set Comprehension

The set comprehension operation is found in SETL, MIRANDA, HASKELL, and other modern functional languages. As demonstrated by the SETL examples in this section, most of the collection operations that are defined on unordered collections can be described with set comprehensions. In this sense, set comprehensions subsume many of the other types of collection operations. They do hoewever have the problem that they cannot easily handle application of functions with more than a single argument.

A typical set comprehension operation looks like:

```
{e(x)  :  x in S | p(x)}.
```

Here S is any set-valued expression, e is a function and p is a boolean predicate. Both e and p are defined on elements of S. The result of this statement is the set of all e(x) with x chosen from all the elements in S that satisfy p. Performing a unary apply-to-each operation is set comprehension without an elimination predicate:

```
(αf X)≡ [f(x)  :  x in X].
```

How can this operation be implemented in terms of the already discussed collection operations on ordered sets? At first glance, set comprehensions looks just like the result of some sort of binding apply-to-each operation involving if:[13]

```
(elwise ((s S)) (if (p s) (e s) (?))).
```

The question mark indicates the problem: what does an application of the if evaluate to when p is not true? Whatever is returned gets placed into the result collection, which would be incorrect. The next operation to try is some sort of pack:

```
(αe (pack S (αp S))).
```

Unfortunately this does not work: since all collections here are unordered, there is no correspondence between the two arguments of the pack. We must use the select generic operator, which ignores the ordering of its arguments:

```
(αe (select p S))
```

is correct. For this same reason, pack cannot implement select for unorderd collections.

The set comprehension notation extends naturally to ordered collections (tuples in SETL) and can be useful in this case as well. Unfortunately there are some problems that may occur. For example, to use set comprehension to specify a binary apply-to-each on two tuples of the same length, one cannot do the following:

---

[13]The careful reader will note that nowhere have we defined the result of an αif. This is intentional. It is a bit too complicated and language specific. Also, the operation creates (solvable, but difficult) problems for massively parallel implementations.

```
[f(x,y):  x in X, y in Y].
```

This produces the value of `f(x,y)` for each ordered pair `[x,y]` in the set product of `X` and `Y`:

```
[x + y :  x in [3, 4], y in [5, 7]]   ⟹ [15, 21, 20, 28].
```

In order to compute the pairwise product of the tuples one must write:

```
[f(X(i), Y(i)) :  i in domain(X)],
```

explicitly using the domain of the sets as indices in the specification. An interesting extension to set comprehensions in SETL would be to allow some notation for implicitly creating a correspondence between tuples (which could be used for binary apply-to-each operations) without resorting to index lists.

5 Language Specific Collection Operations

The previous section described various collection operations from an abstract point of view. We now shift our focus to the specific collection operations supported by several collection-oriented languages: FORTRAN 90, APL, SETL, CM-LISP, and PARALATION LISP. Particular note should be made of the interaction between the collection types provided and the definition of the collection operators.

5.1 Reduction

FORTRAN 90 has intrinsic functions for computing reductions with a fixed set of operators: +, ×, min, max, and, and or reductions are each specified by a keyword. For example, if `A` is an array or vector, then (+-reduce `A`) is denoted by `SUM(A)`. Each of the reduction operations has two optional arguments delineated by the keywords `DIM` and `MASK`. The `DIM` argument is a list of integers indicating the dimensions of the array over which the reduction is to be performed:

$$\texttt{SUM (PRODUCT (A, DIM = 1))} \implies \sum_j \prod_i A_{ij}.$$

`MASK` allows a select operation to be performed on the array before the reduction is carried out:

$$\texttt{SUM (X, MASK = X .GT. 0.0)} \implies \sum_{X_i>0} X_i.$$

`MASK` may be either a boolean valued function or a boolean array comformable with the argument. Each of these optional arguments defaults to allow the reduction of the entire array. If the array being reduced is empty, or if the `MASK` is true for no elements, the identity element is returned. Of the allowed reductions, all are commutative and only floating point `SUM` and `PRODUCT` are not associative. The current draft standard for FORTRAN 90 [28] only specifies that the evaluation of a reduction should produce a "processor-dependent approximation" to the correct value: no explicit order of operation is defined, permitting efficient parallel or vector implementation.

The expression `f/A` is used in APL to specify reduction of an array `A` by a binary function `f`. Standard APL requires `f` to be one of a fixed set of built-in dyadic scalar functions, but the more recent dialects generalize this. The semantics of the reduce operation define the result of reducing a vector by `f` to be equivalent to that of evaluating the expression obtained by writing the vector with an `f` between each two adjacent elements. Since the evaluation of any APL expression is carried out from right to left, the following is an identity:

```
f/1 2 3 4  ≡  1 f 2 f 3 f 4
           ≡  (1 f (2 f (3 f 4)))
```

This gives a well-defined result for non-commutative reduce:

```
-/1 2 3 ≡ (1 - (2 - 3))   ⟹ 2.
```

Unlike FORTRAN 90, APL only permits a single dimension of a multi-dimensional array to be reduced at once. An optional axis argument may be used to select this dimension, which defaults to the last axis:[14]:

```
A          ⟹   3  4  5
               6  7  8
+/[1] A    ⟹   9 11 13
+/[2] A    ⟹   12 21
+/A        ⟹   12 21
```

SETL's reduce operation is very similar to APL's and uses the same syntax. The differences are that there is no restriction to built-in operations, there are no extra dimensions to worry about (since the basic collection type is not grid-ordered), and evaluation proceeds left to right. Also, since SETL's sets are unordered, reduction with a non-commutative function should use tuples to obtain a well-defined result. reduce in SETL permits an optional left argument specifying an element with which to begin the reduction (instead of the identity element). This arguments acts as a default value if the collection is empty:

```
0 +/S
```

gives the sum of the elements of `S∪{0}`.

CM-LISP uses the construct `(βf A)` to indicate reduction of `A` by `f`, where `f` is any CM-LISP function or lambda expression:

```
(β+ '{1→2 2→4 3→4}) ⟹ 10.
```

CM-LISP explicitly guarantees that the order of evaluation of a reduce is undefined for a general xapping: the compiler or interpreter is free to structure the calculation in the most efficient manner. When reducing a xector, the ordering of the xector is respected, yielding a reduction order predictable up to associativity. The reduce is a special case of the more general three argument CM-LISP β (see permute section below).

In PARALATION LISP, reduce is performed by the `vref` function. Given the field `field` of some paralation and a binary function `f`, `(vref field :with f)` computes the reduction

[14]This example assumes that the index origin is set to 1

of `field` with `f`:

```
(vref '#F(1 2 3 4) :with #'+)    ⟹ 10
```

(the #'+ must be used to specify the functional argument + in COMMON LISP). PARALATION LISP allows a default value to be specified for a reduction of an empty field with the optional `:else` keyword. `vref` can be implemented and described in terms of the more general move operator, discussed later (see permute section below).

5.2 Append

In this section we examine mechanisms to append one collection to another. Once, again the definition of this operation is highly dependent on the order type of the collections being combined.

In SETL, two sets or tuples may be appended by using the + operator. In the case of sets, the resulting set is the union of the two arguments (all duplicates are removed):

```
{2, 3, 4, 5} + {4, 5, 6, 7}  ⟹ {2, 3, 4, 5, 6, 7}.
```

With tuples, the + operator just concatenates the two arguments:

```
[2, 3, 4, 5] + [4, 5, 6, 7]  ⟹ [2, 3, 4, 5, 4, 5, 6, 7].
```

If instead of concatenating in this manner one wishes to add an element to a set or a tuple, the `with` statement can be used:

```
{2, 3, 4, 5} with {4, 5, 6, 7}  ⟹ {2, 3, 4, 5, {4, 5, 6, 7}}
[2, 3, 4, 5] with [4, 5, 6, 7]  ⟹ [2, 3, 4, 5, [4, 5, 6, 7]].
```

Each these results can also be obtained by using +, and making the second argument a nested set or tuple; for example,

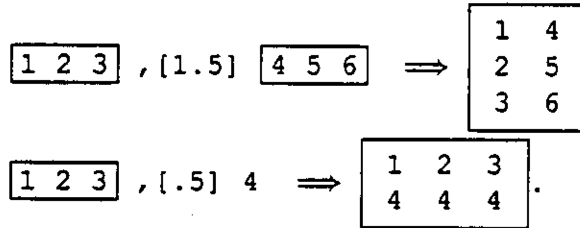
```
{2, 3, 4, 5} + {{4, 5, 6, 7}}  ⟹ {2, 3, 4, 5, {4, 5, 6, 7}}.
```

In APL, there are two types of append operations. The first is for appending elements onto a vector, or for appending one vector to another, and is denoted by the binary , (comma). This operation concatenates (or just *catenates*, in APL lingo) the second argument onto the end of the first:

```
[1 2 3] , [4 5 6]  ⟹ [1 2 3 4 5 6].
```

The other possibility is to join the arguments together "side-by-side"—creating a new array of one greater rank and joining the arguments along the new dimension. This operation is called *lamination*, and is denoted by giving the catenate operator an extra axis specifier argument. This is a fractional value indicating where the new dimension should be added:

```
                          ┌───────┐
[1 2 3] ,[.5] [4 5 6]  ⟹  │ 1 2 3 │
                          │ 4 5 6 │
                          └───────┘
```



The axis specifier of .5 indicates that the new dimension should go before the first dimension (any value between 0 and 1 will work). Similarly, the 1.5 indicates that the new dimension should be after the first dimension. This is readily extended to arbitrarily dimensioned arrays. In the final example, implicit scalar extension changes the 4 into a vector long enough to be laminated.

The append operation in CM-LISP is complicated by the very general nature of the key-ordered collections the language supports. The primitive `xunion` is the provided mechanism for appending xappings. `xunion` takes three arguments: a combining function and two xappings. The index set of the resulting xapping is the union of the index sets of the two argument xappings. If an index occurs in both xappings, the corresponding values are combined by the combining function. `xunion` with an arbitrary combining function is equivalent to append for xappings with disjoint key sets:

```
(xunion #'foo '{1→one 2→two} '{3→three}) ⟹
                    {1→one 2→two 3→three}
```

where `foo` is any function of two variables. If the combining function instead selects its first or second argument, `xunion` is precisely set union:

```
(xunion #'first '{one two} '{two three}) ⟹
                     {one two three}
```

where `first` returns the first of its two arguments. Appending one xector to another presents a problem. A xector is just syntax for a xapping whose index set is the first n natural numbers, where n is the length of the vector. Thus, any two non-empty vectors are going to have intersecting index sets (the smaller set will be a subset of the larger). To perform an append, all elements in the index set of the second vector must be incremented by the length of the first vector. Then `xunion` can be applied. This can be accomplished with a β operation, but the resulting function uses features of the language that we do not discuss in this paper. For the general case of appending two key-ordered collections whose keys are not disjoint, it is not even clear what result is naturally expected.

As with CM-LISP, the operation of appending two fields in PARALATION LISP is not as simple as just concatenating the values together: a new paralation must be created in which to hold the results. The primitive function for append is `field-append-2`. This function takes two fields, which may be from any (different) paralations, and returns a new field, in a new unshaped paralation. This field contains the concatenated contents of the two argument fields:

```
(field-append-2 #F(a b c) #F(d e)) ⟹ #F(a b c d e).
```

A second command, expand is provided which concatenates all the field valued elements of a field together into a single field of a new paralation:

```
(expand #F(#F(a b c) #F(d e))) ⟹ #F(a b c d e).
```

expand can be implemented using reduce and field-append-2:

```
(defun expand (field)
      (vref field :with #'field-append-2
                  :else (make-paralation 0)))
```

5.3 Permute

FORTRAN 90, APL and SETL each implement permute as one form of a generalized subscripting mechanism. In contrast, both CM-LISP and PARALATION LISP couch permute in terms of general communication primitives. This section examines both subscripting and communication in detail.

FORTRAN 90 has a wide range of methods for indexing arrays. An index into an array is a tuple whose length is the rank of the array; it can also be a vector of such tuples. Each element of the tuple can be a scalar or a triple of the form start:end:stride. For example, suppose A has been declared to be of type INTEGER A (10,5):

```
A(3,4)       ⟹ A(3,4)
A(1 2, 2 5)  ⟹ | A(1,2)  A(1,5) |
                | A(2,2)  A(2,5) |
A(2:7:2, 3)  ⟹ A(2,3), A(4,3) A(6,3).
```

An inverse-permute can be performed by creating a permutation vector and indexing into the array to be permuted. These general indices may also appear on the left hand side of an assignment, thereby permitting selective assignment to portions of an array. The only restriction (to allow vector and parallel implementation) is that no array location is selected more than once. A regular permute is accomplished by using the permutation vector to specify assignment into the array: (if B is an array of size 4)

```
B(3 4 2 1) = (/ 8 9 10 11 /)
B  ⟹  (/ 10 11 9 8 /).
```

APL has its own version of generalized indexing. In APL, any part of an indexing subscript may be a scalar, vector or array. The shape of the resulting array is equal to the catenation of the shapes of each subscript. For any vector component, all the matching elements of the array will be chosen. An empty element indicates that the entire column should be selected. As with FORTRAN 90, if these indices are used on the left-hand side of an assignment only those selected elements of the array are assigned. APL allows subscripts on the left-hand side to be repeated, but the outcome of the assignment is dependent on the implementation. An inverse-permute is done by indexing into the data with the appropriate subscripts:

```
A ← 1 2 3 4 5
A[3 4 2 1 5]  ⟹ 3 4 2 1 5
```

Permuting elements of a higher dimensional array cannot be done in this manner since too many indices will match the list:

```
A ← 2 3 ρ ι6      ⟹   1 2 3
                       4 5 6

A[1 2 ; 3 1]      ⟹   3 1
                       6 4
```

This does not select only `A[1;3]` and `A[2;1]`. However, note that these are the diagonal elements in the array produced; this will be true in the general case. APL provides a mechanism (dyadic transpose) to pull out the generalized diagonal of a multidimensional array.

The permute operation on tuples in SETL is expressed in terms of indexing and set comprehension:

```
A := [2 4 6 8]
[A[i] :  i in [3 1 4 2]]   ⟹  [6 2 8 4].
```

There is no generalized indexing facility as in FORTRAN 90 or APL. Indices can only be scalars or intervals; sets and tuples are not permitted.

CM-LISP has a very general form of permute that arises quite naturally from the data-parallel origins of the language. CM-LISP is intended for massively parallel machines with a very large number of processors and arbitrary communication facilities. It is expected that every element of a xapping is governed by a different (virtual) processor. In this model, a permute may be thought of as a primitive for processor communication: a method to send information from one processor to another. The natural question to ask about such a primitive is "What happens when more than one piece of data is sent to the same processor?" CM-LISP answers this by combining the colliding data in some manner, just as with the `xunion` function described above. CM-LISP has a primitive β operator used to describe communication. This operator takes three arguments: a combining function and two xappings. The result of this operation is fully described in Section 4.1.2. This permute also allows the computation of a key-ordered reduce: `(βf d x)` computes the xapping whose indices come from the values of xapping `d` and whose values are the `f` reduction of values of xapping `x` with those same indices:

```
(β+ '{ 1→2 2→4 3→4 } '{ 1→5 2→6 3→7 })  ⟹  { 2→5 4→13 }
```

PARALATION LISP, like CM-LISP, also uses permute for specifying interprocessor communication. The basic primitive for all data movement is the `<-` (move) function. This command takes up to four arguments: a source field, a combining function, a default field and a mapping. A *mapping* is a fixed communication pattern and can be created by the `match` command. `match` takes two fields (`to-field` and `from-field`) as arguments and returns a communication pattern. This pattern connects a site in `from-field`'s paralation with a site in the `to-field`'s paralation if and only if the values in the key fields of these paralations are equal. The `<-` works as follows: data is transferred according to the mapping from the source field to a new field in the destination paralation. Any collisions are reconciled with the combining function, and the default field fills all the gaps. Here is a simple example that computes the product of the elements in `primes` that are also in `nums`:

```
(setq nums (make-paralation 8))   ⟹  #F(0 1 2 3 4 5 6 7)
(setq primes '#F(2 3 5 7 11 13))
(setq lil-primes
```

```
(<- primes :by (match primes nums)
           :default (elwise ((num)) '1)))
                                ⟹ #F(1 1 2 3 1 5 1 7)
(<- lil-primes :with #'*
               :by (match (make-paralation 1)
                          (elwise (lil-primes) 0)))   ⟹ 210
```

The first <- creates a field of all the elements in `primes` that are also in `num`, with 1 as the default. The second <- creates a mapping that forces all sites of a field in the `lil-primes` paralation to be sent to the same place–the one site of `(make-paralation 1)`. These are then combined with multiply. The `vref` command can be implemented in terms of <- using this same technique.

Pack and Select

The pack and select operations both involve pulling particular elements out of a collection and putting them into a new collection. They are examined jointly in the section.

FORTRAN 90 has a primitive `pack` command that packs arrays into vectors. The arguments to `pack` are an array, a boolean mask (which may be a constant or a predicate) conformable with the array, and an optional vector that specifies both the mimimum length of and default values for the result:

```
        | 1  2  3 |
M  ⟹    | 4  5  6 |
PACK(M, M .GE. 3)  ⟹ (/3, 4, 5, 6/)
PACK(M, M .GE. 5, VECTOR = (/0, 0, 0, 0/))  ⟹ (/5, 6, 0, 0/).
```

The APL version of pack is called compress and is denoted by `mask/value`, where `mask` is a boolean array whose shape can be extended to conform with `value`:

```
1 0 1 0 1 / 1 2 3 4 5  ⟹ 1 3 5
1 / 1 2 3 4 5  ⟹ 1 2 3 4 5
          | 1  2  3 |       | 1  3 |
1 0 1 /   | 4  5  6 |   ⟹   | 4  6 |.
```

To perform a select, a boolean mask can be created that can be used for a compress operation:

```
(A > 3) / A.
```

pack and select in SETL are described as simple set comprehensions:

```
{x : x in A | P(x)}
```

extracts all those elements of `A` that satisfy the boolean function `P`. This is discussed in Section 4.2.3

pack requires the movement of data and the creation of a new paralation in PARALATION LISP. As such, it must be implemented with <-. PARALATION LISP provides the `choose` function for creating a new paralation and a mapping between this paralation and the true

elements of a boolean field of another paralation. This allows data to be moved from a field of the original paralation to the new paralation:

```
(setq a (make-paralation 5))  ⟹ #F(0 1 2 3 4)
(setq mask (elwise ((a)) (prime? a)))  ⟹ #F(NIL NIL T T NIL)
(<- a :by (choose mask))  ⟹ #F(2 3).
```

This may look clumsy, but the only extra operation that is performed is the creation of the mapping from the mask. This mapping may be stored and then used later to move data in other fields of the paralation with only a <-.

6 Conclusion

This paper has reviewed and compared a set of languages we call "collection-oriented". Although these languages originated independently of massively parallel machines, they have had a recent revival with the advent of such machines. The collection operations of the languages are naturally parallel and map well to existing parallel architectures. The amount of parallelism inherent in these operation is much greater and more easily accessible than the parallelism available from other programming methodologies.

The paper separated the discussion of the *collection types* and of the *collection operations* on these types. It showed how collection types can differ in how they are ordered, in whether the elements of a single collection must all be of the same type, and in whether a collection can be nested. The paper described two classes of collection operations: *aggregate* operations and *apply-to-each* forms. The aggregate operations of a language operate on a collection as a whole, such as rearranging or summing the elements, while the apply-to-each forms map some function over the elements of a collection, such as calculating the square-root of every element.

The work on mapping the most interesting of the collection-oriented languages onto massively parallel machines has just began, and we believe that this will be a very important area of research in the next decade.

Acknowledgements

We wish to thank Gary Sabot and Skef Wholey for their conversations and ideas regarding the contents of this paper.

References

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

[2] K. E. Batcher. *The Massively Parallel Processor System Overview*, pages 142–149. MIT Press, Cambridge, MA, 1985.

[3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

[4] Timothy A. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[5] T. Busse. MPP Pascal. In *The 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 595–9, 1988.

[6] Wai-Mee Ching. Program analysis and code generation in an APL/370 compiler. *IBM J. Res. Develop.*, 30(6):594–602, November 1986.

[7] Janice Glasgow, Michael Jenkins, Carl McCrosky, and Henk Meijer. Expressig parallel algorithms in Nial. *Parallel Computing 11*, pages 331–347, 1989.

[8] R. Greenlaw and L. Snyder. Achieving speedups for APL on an SIMD parallel computer. *APL Quote Quad*, 18(4):3–8, June 1988.

[9] W. Daniel Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

[10] Paul Hudak and Philip Wadler. Report on the functional programming language HASKELL. Technical Report 1.0, Yale University, New Haven, April 1990.

[11] IBM. *APL2 Programming: Language Reference*, first edition, August 1984. Order Number SH20-9227-0.

[12] Kenneth E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[13] Kenneth E. Iverson. A dictionary of APL. *APL Quote Quad*, 18(1):5–40, September 1987.

[14] M.A. Jenkins, J.I. Glasgow, and C. McCrosky. Programming styles in NIAL. *IEEE Software Engineering*, January 1986.

[15] Clifford Lasser. The essential *Lisp manual. Technical report, Thinking Machines Corporation, Cambridge, MA, July 1986.

[16] Trenchard More. The nested rectangular array as a model of data. In *APL 79 Conference Proceedings*, pages 55–73. ACM, 1979.

[17] D. Parkinson, D. J. Hunt, and K. S. MacQueen. The AMT DAP 500. In *Spring COMPCON 88; digest of papers*, pages 196–197. The Computer Society of the IEEE, IEEE Computer Society Press, 1988.

[18] John Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL87-5, Thinking Machines Corporation, April 1987.

[19] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, Cambridge, Massachusetts, 1988.

[20] J. T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[21] Guy L. Steele Jr. CM-Lisp. Technical report, Thinking Machines Corporation, 1986.

[22] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common LISP: The Language*. Digital Press, Burlington, MA, 1984.

[23] Gerald Jay Sussman and Jr. Guy Lewis Steele. Scheme: an interpreter for extended lambda calculus. Artificial Intelligence Laboratory, Massachusetts Institute of Technology 349, Massachusetts Institute of Technology, Cambridge, MA, December 1975.

[24] Thinking Machines Corporation. Model CM-2 technical summary. Technical Report HA87-4, Thinking Machines Corporation, Cambridge, Massachusetts, April 1987.

[25] Hai-Chen Tu and Alan J. Perlis. FAC: A functional APL language. *IEEE Software*, pages 36–45, January 1986.

[26] David Turner. An overview of MIRANDA. *SIGPLAN Notices*, December 1986.

[27] C. Walinsky and D. Banerjee. A functional programming language compiler for massively parallel computers. In *ACM Conference on Lisp and Functional Programming*, pages 131–138, 1990.

[28] X3J3. *ANSI Fortran Draft S8, Version 111*. ANSI.