GENERAL ELECTRIC

General Electric Company
Corporate Research and Development
Schenectady, New York 12345

TECHNICAL INFORMATION SERIES

AUTHOR Kapur, D
Musser, DR
Stepanov, AA

SUBJECT functional programming

NO. 81CRD114

DATE August 1981

TITLE
Operators and Algebraic Structures

GE CLASS 1

NO. PAGES 13

ORIGINATING COMPONENT Automation and Control Laboratory

CORPORATE RESEARCH AND DEVELOPMENT SCHENECTADY, N.Y.

SUMMARY

Operators in functional languages such as APL and FFP are a useful programming concept. However, this concept cannot be fully exploited in these languages because of certain constraints. It is proposed that an operator should be associated with a structure having the algebraic properties on which the operator's behavior depends. This is illustrated by introducing a language that provides mechanisms for defining structures and operators on them. Using this language, it is possible to describe algorithms abstractly, thus emphasizing the algebraic properties on which the algorithms depend. The role that formal representation of mathematical knowledge can play in the development of programs is illustrated through an example. An approach for associating complexity measures with a structure and operators is also suggested. This approach is useful in analyzing the complexity of algorithms in an abstract setting.

KEY WORDS

algebraic structures, operators, functional programming, functional languages, algorithm design, algorithmic complexity, verification, specification

INFORMATION PREPARED FOR _____

_____

Additional Hard or Microfiche Copies Available from

Technical Information Exchange
Bldg. 81 Room A133, Schenectady, N.Y. 12345

OPERATORS AND ALGEBRAIC STRUCTURES

D. Kapur, D.R. Musser, and A.A. Stepanov

1. INTRODUCTION

One of the most important notions which makes functional languages different from conventional programming languages is the notion of operator or functional form. Along with a notion of abstract data types, it constitutes the most interesting development in programming languages since the early sixties. We are going to make some suggestions about further development of these two notions and their possible merger. Our main thesis will be that operators should be defined within the context of the algebraic structures to which they naturally belong. We shall describe a fragment of a programming language that supports the description of algebraic structures and their operators, and illustrate the role that formal representation of mathematical knowledge can play in the development of programs.

Operators were systematically introduced into programming by Kenneth Iverson.[7] His APL language is a widely used programming language which contains a rich set of operators. According to Iverson, "an operator is an object which applies to a function or functions to produce a related function" (Ref. 7, p. 161). However, the use of operators in APL is limited by the fact that it is impossible to apply operators to user-defined functions.

Let us consider *reduction,* a commonly used operator in APL. It is monadic, which means that it takes one function as an argument. The reduction operator applied to the function "plus", for instance, produces the "summation" function. Reduction takes as its argument any primitive binary scalar function. Since user-defined functions do not have types, the APL system cannot distinguish between scalar and vector user-defined functions. Another property of the function which the reduction operator must know is the existence of left and right identity elements (Ref. 4, p. 19). Yet there is no way to specify such elements for user-defined functions in APL.

Another problem with operators in APL is that it is impossible for the user to define new operators, since APL functions do not take functions as their arguments.

John Backus in his Formal System for Functional Programming (FFP) attempted to resolve both of these problems.[2] The problem of defining the types of user-defined functions is eliminated because there is no typing in FFP. Any function can be applied to any element of the universal domain of objects. The problem of specifying properties of the user-defined functions still remains. For example, the "insert" functional form, which is the FFP analog of the APL reduction operator, utilizes the existence of a unique right identity element, but there is no facility in the language for defining the right identity of a function.

The problem of defining a new functional form is resolved by representing functions as objects of the same universal domain, and then defining new functional forms as functions which operate on representations of other functions. But by doing so, FFP loses one of its novel

Manuscript received May 4, 1981.

features: association between functional forms and algebraic laws for them. It is impossible to specify laws associated with a user-defined functional form (Ref. 2, p. 633).

If we consider the use of reduction operator in APL, it will become apparent that in many cases, such as "plus", "times", "and", "or", "minimum", and "maximum", the result is independent of the order in which the reduction is performed. This property of the reduction operator applied to these functions becomes important for parallel computers. We shall call this case of the reduction operator the *parallel* reduction operator. As it will be shown later, even in the case of sequential computation it can provide a basis for organizing programs and obtaining more efficient algorithms. All functions for which the reduction can be executed in an arbitrary order share the same two properties: associativity and commutativity. An algebraic structure with an associative and commutative operation is called a commutative semigroup. So the parallel reduction operator is applicable in the algebraic structure of a commutative semigroup.

A structure on a finite family of sets is defined as a finite number of operations satifying a system of axioms. If any two structures satisfying a system of axioms are isomorphic, then the theory generated by the system of axioms is said to be *univalent;* otherwise, the theory is said to be *multivalent.* We will also call structures satisfying a univalent theory as univalent structures and structures satisfying a a multivalent theory as multivalent structures. For example, structures of "integers", "real numbers", and "group of order 3" are univalent. On the other hand, structures of "semigroup", "ring", and "group of order 4" are multivalent (Ref. 1 p. 385). (If we were to restrict ourselves to first order theories, "real number," for example, would be multivalent as non-standard models exist; however, we make no such restriction. The theorem that "any two completely ordered fields are isomorphic" may be found in any advanced calculus text.) We have decided to use the old-fashioned formalism of Bourbaki instead of the category theoretic formalism used by Burstall and Goguen,[3] because we believe that it is much less esoteric.

On any structure we can define new operations with the help of the set of primitive operations. In the case of univalent structures, these new operations correspond to user-defined functions in APL or FFP. But in the case of multivalent structures, they correspond to operators or functional forms. It is, thus, reasonable to incorporate the notion of structure into the framework of functional languages.

In the next section, we shall build a language to illustrate how it can be done. We will describe it informally, omitting many details and illustrating undefined metavariables with examples. This language is a part of a very high-level language *Tecton* ("builder") for describing software systems which is being developed at the General Electric Research and Development Center.

Before continuing, we remark that the subject of structures in programming languages is under active investigation by a number of other authors. We note particularly the work of Guttag,[6] Zilles,[13] and the ADJ group[5] on abstract data types, of Burstall and Goguen[3] and Nakajima et al.[11] on hierarchical specification languages, and the efforts to formally describe computer algebra systems by Jenks,[8] Winkler,[12] and Zippel.[14]

2. STRUCTURES IN TECTON

The description of a structure includes its *configuration,* which is the list of structures from which the described structure is built and the list of primitive operations defined on these structures. Examples of configurations are:

group(S:set; + :S+S → S, inv:S → S, 0: → S)

module(G:abelian group, R:ring; *:R*G → G)

The first member of the list of structures is called the "base structure" of the configuration (G is the base structure of the configuration for module). In the list of operations we write *:R*G → G to mean that * is a binary infix operation whose domain is $R \times G$ and whose range is G. (If * were to be used as a prefix operation, we would write *:R,G → G.)

A structure description may also include a set of axioms and theorems known about the structure and a set of secondary operators, defined in terms of the primitive ones. The axioms and theorems may include a description of properties of the complexity of primitive and secondary operations, as will be discussed in Section 4.

These parts of the description are introduced by constructs of Tecton which permit creating new structures from existing ones and modifying existing structures. The most important are: *create, enrich, inform, provide, instantiate, implement,* and *represent.*

The *create* construct adds a new element to the domain of structures. Its format is:

create <configuration> [with <properties>]

(Brackets enclose parts of the construct that can be omitted.) For example, if we have a structure "set", then we can add a structure "group":

create group(S:set; +:S+S → S, inv:S → S, 0: → S)
with associativity: $x+(y+z) = (x+y)+z$,
leftidentity: $0+x = x$,
leftinverses: $\text{inv}(x)+x = 0$;

By convention, the symbols x,y, and z that appear in the properties are variables that are of the type of the base structure (S in this case). When variables of other types are needed, they will be explicitly typed. The names of the properties can be omitted; they merely provide an extra way of referring to the properties.

The *enrich* construct allows us to add a new structure to the domain of structures by means of adding new axioms to an existing structure. Its format is:

enrich <structure name> [into <structure name>] with <properties>

For example, having created "group" we can introduce a structure "abelian group":

enrich group into abelian group
with $x+y = y+x$;

As another example, a "group with all elements of order 2" can be introduced by:

enrich group

with all elements of order 2: $x+x = 0$;

The *instantiate* construct replaces the formal parameters of a configuration with actual parameters, giving an instance of one structure within another. Its format is:

instantiate <structure name> of <structure name>
[as <name>][<plugged interface>]

For example, if, in addition to the previously defined structures, we have in the domain of structures the structure of "integers" with the configuration

integers(I:set; $+$:$I+I \rightarrow I$, negate:$I \rightarrow I$, 0:$\rightarrow I$)

then we can

instantiate abelian group of integers

($S = I$, $+ = +$, inv = negate, $0 = 0$);

We did not give any name to the structure we just created, so we can reference it only as "abelian group of integers".

The *inform* construct allows us to add new theorems to an existing structure. Its format is:

inform <structure name> that <properties>

For example, we can add some useful properties to the structure of "group":

inform group that $x+0 = x$, $x+\text{inv}(x) = 0$;

Note that this does not produce a new structure, since these properties are provable from the axioms of "group".

The *provide* construct allows us to define additional operators on the already defined structures. Its format is:

provide <name> with <operator definitions>

For example, we can now define a subtraction operator on the "group" structure:

provide group with $-$: $x-y \rightarrow x+\text{inv}(y)$;

From now on we can use the subtraction operation with any instance of "group".

The *implement* construct allows us to specify some special ways in which operators can be implemented on those instances of a structure which possess some particular set of properties. Its format is:

implement <operator name> [on<structure name>] [with <properties>]
as <implementation>

For example, on groups with all elements of order 2, it is possible to simplify the subtraction operator:

implement − on group with all elements of order 2
as $x+y$;

In this example we used a property of the structure, but another possibility is to refer to a property of the particular inputs to the operator, as will be illustrated in Section 4.

The final construct of Tecton we shall discuss is the *represent* construct:

represent <structure name> as <structure name>
using <representation function>
and <abstraction function>

This allows introduction of a mapping from one structure to another as an aid to defining operations or expressing their implementations. The abstraction function is required to be a homomorphism from the second structure back to the first, and is included as a way of guaranteeing that operations on the second structure preserve those of the first. For example, if we have a structure of complex numbers we can introduce their polar representation, which is useful for many algorithms.

We have given only a sketch of some of the main constructs describing structures in Tecton. We shall now attempt to illustrate the main ideas with an example.

3. PROGRAMMING USING STRUCTURES

The first observation which can be made is that it is impossible to program in Tecton as we have described it, since it does not contain any primitive structures from which other structures can be constructed. As an initial set of structures for our exercise in programming, we shall use the structures "set", "multiset", and "sequence" with many different operations and operators defined on them, which will be seen in context. As a theme for the exercise, we shall select one of the most classical of all programming problems: sorting.

First, we need the notion of a totally ordered set:

create ordered set(S:set; relation $\leqslant$: $S \leqslant S$)
with $x \leqslant x$,
$x \leqslant y$ and $y \leqslant z$ implies $x \leqslant z$,
$x \leqslant y$ and $y \leqslant x$ implies $x=y$;

enrich ordered set into totally ordered set
with $x \leqslant y$ or $y \leqslant x$;

Then we need to have something to sort:

create orderable sequences (sequences of totally ordered set);

and a way to distinguish sorted sequences:

create ordered sequences (OrdSeq:subset of orderable sequences
such that for all u and for all x,y in u,
x precedes y in u if and only if $x \leqslant y$).

Now we can write our first program:

provide ordered sequences with
merge:$x,y \rightarrow$
if x = null or y = null then x cat y
else if head(x) $\leqslant$ head(y)
then <head(x)> cat merge(tail(x),y)
else <head(y)> cat merge(x,tail(y));

It is easy to see that the null sequence is an identity element for the merge function and that merge is both commutative and associative. Thus we may:

inform ordered sequences that
merge(null,x) = x = merge(x,null),
merge(x,y) = merge(y,x),
merge(merge(x,y),z) = merge(x,merge(y,z));

This makes it reasonable to make use of some additional structures:

create semigroup(S:set; $+:S+S \rightarrow S$)
with associativity: $x+(y+z) = (x+y)+z$;

create monoid(S:semigroup; $0: \rightarrow S$)
with $0+x = x+0 = x$;

enrich monoid into abelian monoid
with commutativity: $x+y = y+x$;

and to introduce an operator reduction:

provide sequences of monoid with
reduction: $x \rightarrow$ if x = null then 0
else head(x) + reduction(tail(x));

Thus there is an instance of an abelian monoid in ordered sequences:

instantiate abelian monoid of ordered sequences
as merge(S = OrdSeq, + = merge, 0 = null)

Here we have used the function name "merge" also as the name of the instantiated structure. We shall refer to it as the "merge monoid". When we use an operator such as "reduction"

defined in the monoid structure, we will denote the corresponding operator in the merge monoid as "reduction of merge". Thus we can use "reduction of merge" on sequences of ordered sequences to merge them into one.

The next step is to recognize that sequences under "cat" also form a monoid:

inform sequences that
null cat x = x cat null = x,
(x cat (y cat z)) = ((x cat y) cat z);

instantiate monoid of sequences
as cat(S = sequences, + = cat, 0 = null);

Our purpose with this step is to define an abstraction function for the following representation.

represent orderable sequences as sequences of ordered sequences
using seqrep
and seqabs: $x \rightarrow$ reduction of cat(x);

We have only given the representation function a name, "seqrep", without defining it. The reason for this will be seen in a moment, but now we can write our program for sorting as

provide orderable sequences with
sort: $x \rightarrow$ reduction of merge(seqrep(x));

This is a generic algorithm for two reasons. One is that different implementations of the reduction operator will give different algorithms, a point we shall study in the next section. The other reason is that we can instantiate "seqrep" in various ways. One possibility is:

provide sequences with
oneify: $x \rightarrow$
if x = null then null
else <<head(x)>> cat oneify(tail(x));

instantiate sort as sort1(seqrep=oneify);

Thus oneify($<a_1, a_2, \ldots, a_n>$) = $<<a_1>, <a_2>, \ldots, <a_n>>$. Another possibility would be a function "runify" that keeps runs of increasing elements in the same sequence; e.g.,

runify($<3,5,1,8,9,11,4,3,8>$) = $<<3,5>, <1,8,9,11>, <4>, <3,8>>$.

This would be useful when it is known that long runs are likely to exist, i.e., when the input to "sort" is known to be almost sorted.

4. ALGEBRAIC OPTIMIZATION

As we have thus far defined "sort", it is not very efficient. Because of the way we defined the reduction operator to perform leftmost reductions, we are using the merge operation on ordered sequences merely to do insertions of a single element into an ordered sequence. Thus the algorithm we have obtained is essentially "insertion sort", an order n^2 algorithm. Let us now see how a simple redefinition of reduction has the consequence that our already given sort program becomes an order $(n \log n)$ algorithm.

In making this new definition of reduction, we do not want to simply replace the current definition, since the current definition may in fact be more efficient in other applications. The key idea we want to demonstrate now is that more than one definition of an operator can exist and the decision as to which one to use in a particular application should be made on the basis of complexity properties of the structures involved in the application.

For studying the complexity of the implementations of secondary operators associated with a structure, it is necessary to specify complexity information with the primitive operations and the base structure in its configuration. Every structure is assumed to have an implicit real-valued nonnegative function *length* defined on the elements of the base structure of its configuration, and each operation, operator, and implementation of an operator is assumed to have an implicit real-valued nonnegative function *cost* associated with it. The *enrich* and *inform* operations can be used to add axioms and theorems expressing complexity properties in terms of *length* and *cost*.

The discussion of the complexity analysis in this report will be very sketchy because of the scope of this report. We will only introduce a few concepts to illustrate some ideas in comparing different implementations of the reduction operator.

Our first task is to add some complexity properties to abelian monoids.

```
enrich abelian monoid into Huffman monoid with
    length(x+y) = length(x) + length(y)
    cost of + of (x,y) = order(length(x) + length(y));
```

The name of this structure derives from D. Huffman's algorithm (Ref. 9, Vol. 1, pp. 402-405 and Vol. 3, p. 365) for finding a tree with minimum weighted path length. With this definition, it is possible to show that the merge monoid can be made into a Huffman monoid.

```
inform ordered sequences that
    additive length: length(merge(x,y)) = length(x) + length(y)
    linear cost: cost of merge of (x,y) = order(length(x) + length(y))
```

Now it makes sense to

```
implement reduction on sequences of Huffman monoid as
    huff(makemultiset(x))
```

where "huff" is an operation on multisets that, as in Huffman's algorithm, chooses a pair of elements to be combined based on minimality of their length:

```
provide multisets with
    huff: s → if s = empty then 0
                else if singleton?(s) then u where u : s
                    else huff( (s − {u,v}) ∪ {u+v})
                        where u,v: s and minimalLength(u,s)
                            and minimalLength(v,s−{v});
```

We now see that, since the merge monoid is a Huffman monoid, our sort program will be able to use this implementation. Thus it becomes the well-known merge sort algorithm, whose cost is order ($n \log n$).

With an additional property of the input, we can simplify the Huffman implementation of reduction.

```
provide sequences with
    relation equalLengths: x → for all u,v in x, length(u) = length(v);

provide sequences of Huffman monoid with
    huff1: x → if x = null then 0
          else if singleton?(x) then head(x)
              else huff1( tail(tail(x))
                  cat <head(x) + head(tail(x))> )

implement reduction on sequences of Huffman monoid
    with equalLengths(x) as huff1(x);
```

This implementation would be used by sort1, since the instance "oneify" of "seqrep" produces sequences satisfying the "equalLengths" relation.

In closing this discussion of optimization, let us now consider briefly how the reduction operator introduced in Section 2 relates to the parallel reduction operator discussed in the Introduction. In Section 2, the definition of reduction is given recursively in terms of primitive operations "head" and "tail" on sequences. This permits a simple definition, but also implies a commitment to sequential computation. With a different set of primitive operations on sequences we can express the definition of reduction in a way that naturally implies parallel computation. We will thereby obtain the possibility of parallel computation in all applications of reduction that obey the necessary algebraic laws, such as our sort program.

Let us suppose that operations on sequences of monoid include an operator "pairs" that takes a sequence $<x_1, \ldots, x_n>$ into a sequence of two element sequences $<<x_1,x_2>, <x_3,x_4>, \ldots>$, where the last pair is $<x_n,0>$ if n is odd; and that there is a primitive operator "mapall" that applies a function "f" to each element of a sequence, producing the sequence of the results. We could, of course, define these with Tecton, e.g.,

```
provide sequences of domain of function f with
    mapall: x → if x = null then null
          else <f(head(x)) > cat mapall(tail(x));
```

but instead we assume "mapall" is already implemented in a way that permits parallel computation, so that it may be used to implement other parallel operators. For example, we may now

```
provide sequences of monoid with
    addpair: x → head(x) + head(tail(x)),
    parallelReduction: x → if x = null then 0
                else if singleton?(x) then head(x)
                    else parallelReduction (mapall of addpair(pairs(x)));
```

If we now

```
implement reduction as parallelReduction;
```

we obtain a parallel version of any algorithm that uses reduction, such as our "sort" program. We note that "mapall" could also have been used to define the function "oneify", which was defined recursively in the previous section.

5. CONCLUSION

By combining the notion of operators as used in languages such as APL and FFP with the ideas of algebraic structures, we have proposed mechanisms to define structures and associate operators with structures in a functional setting. This allows us to describe algorithms abstractly and without committing to any particular model of computation, thus emphasizing the algebraic properties they depend on for their functional behavior. We have also suggested an abstract way to associate complexity measures with a structure and its operators. Below, we briefly discuss some topics closely related to the ideas presented in the paper which need further investigation.

We used the structures "set", "multiset", and "sequences" for illustrating various language constructs. There is a need to identify other structures useful in describing systems and develop their theory. Like the reduction operator on a monoid, other operators on a monoid and other algebraic structures like group, semi-ring, ring, etc., should be investigated within the proposed language framework. We believe that the proposed language constructs, when used with a library of judiciously chosen structures and operators, can be highly expressive and useful in describing complex systems.

An important topic not discussed in the paper is the role of computer-assisted theorem proving, in relating various structures and operators and deriving properties about structures and operators, as well as about their complexity. An example is for the computer to assist in checking that the monoid structure can be instantiated into sequences by associating the configuration of the monoid with that of sequences and by deducing the monoid properties from the axioms and theorems of sequences. Another example is to prove, using the computer, that the merge monoid discussed in Section 4 is a Huffman monoid by showing that the merge operator is length additive and its cost function is linear. Deducing such information can help in developing efficient implementations of the algorithms. We also need to identify problem domain-independent properties of structures such as univalency, multivalency, consistency,

completeness of axioms, etc., and develop algorithms for checking these properties. In the study of these questions, we will draw heavily upon our experience with the capabilities of the AFFIRM system[9] for theorem proving and analysis of algebraic specifications.

Without giving any details, we have alluded to an abstract way of associating complexity with structures and operators. This approach also seems to provide a unified framework for discussing complexity in both a parallel and a sequential environment. However, much work needs to be done toward developing such an approach as a basis for constructing new algorithms and analyzing their complexity.

One of the main considerations for the design of Tecton is to identify abstraction mechanisms that aid in describing systems in a natural way. The abstraction mechanisms should also be amenable to formal reasoning so that the computer can assist in applying them. In this report, we have introduced several constructs for communicating knowledge of algebraic structures in a way that facilitates the development and selection of algorithms. Besides these constructs, Tecton has constructs for manipulating objects other than structures, which will be discussed in forthcoming reports.

ACKNOWLEDGMENTS

We would like to thank John Guttag, Chuck Fiduccia, and Jim Thatcher for many valuable comments on the first draft of this paper. We would also like to acknowledge John Hutchison's participation in the initial stage of this research.

REFERENCES

1. Bourbaki, N., *Theory of Sets*, Chapter IV, "Structures" and Summary of Results, Section 8, "Scales of Sets. Structures," Addison-Wesley, 1968.
2. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM 8* (21), August 1978.
3. Burstall, R.M., Goguen, J.A., "Putting Theories Together to Make Specifications," Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, August 1977.
4. Falkoff, A.D. and Orth, D.L., "Development of an APL Standard," RC 7542, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, February 1979.
5. Goguen, J.A., Thatcher, J.W., Wagner, E.W., "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types" in *Current Trends in Programming Methodology,* Vol. IV, Data Structuring (R.T. Yeh, ed.), Prentice Hall, Englewood Cliffs, NJ, 1978.
6. Guttag, J.V., "Abstract Data Types and the Development of Data Structures," *CACM 20* (6), pp. 396-404, June 1977..
7. Iverson, K.E., "Operators," *TOPLAS 1*(2), October 1979.
8. Jenks, R.D., Trager, B.M., "A Language for Computational Algebra," *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation,* Snowbird, August 1981.
9. Knuth, D.E. *The Art of Computer Programming*, Vol. 1, Vol. 3, Addison-Wesley, 1968, 1973.
10. Musser, D.R., "Abstract Data Types in the AFFIRM System," *IEEE TSE 1*(6), January 1980.
11. Nakajima, R., Nakahara, H., Honda, M., "Hierarchical Program Specification and Verification - A Many Sorted Logical Approach," preprint RIMS 256, November 1978.
12. Winkler, F., "A Language for Specifying Algebraic Structures," unpublished manuscript, Fall 1979.
13. Zilles, S.N., "An Introduction to Data Algebra," Draft Working Paper, IBM San Jose Research Laboratory, September 1975.
14. Zippel, R., private communication, March 1981.