AT&T Bell Laboratories

Whippany Road
Whippany, New Jersey 07981-0903
201 386-3000

July 22, 1987

University of Waterloo
Technical Report Distribution
Dept. of Computer Science
Ontario N2L 3G1
Canada

To whom it may concern,

To the best of our knowledge the document(s) listed below is(are) not available from the National Technical Information Service or the Defense Technical Information Center. Will you please furnish us with a copy of each, or if not available through your organization, please forward any information you may have regarding the document(s) and a source of supply.

Report CS-80-47
Constructive Approach to the Design of Algorithms and Their Data Structures.
Gonnet, G.H. and Tompa, F.W.
1980

Thank you for your attention in this matter.

sent + invoice for $2.00

Very truly yours,

Cindy L. Anderson

Cindy L. Anderson
Technical Report Specialist
Room 5E-227

(201) 386-6734

In Reply Refer To:

Ref. No. 87-0708-4

received payment

SEP 23 1987

AT&T Bell Laboratories
Technical Report Service
Whippany, NJ 07981

Attn: Cindy Anderson Ref. No. 87-0708-4

The material requested is:

✓ Enclosed

_____ For retention

_____ On loan until __________

_____ Priced at *$2.00 Canadian*

_____ Not available from us but may be obtained from:

Received C. Gordon 8/19/87

Additional identifying information for the requested document is shown below:

Please make your cheque or money order payable to the University of Waterloo, Computer Science Department and forward to my attention.

Thank you.

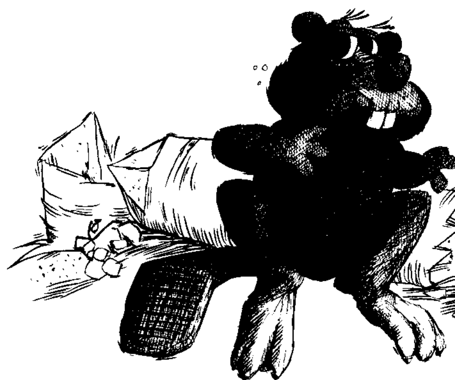
Signed *Susan DeAngelis*

Title *Research Report Secretary*

University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
S. DeAngelis
Research Report Secretary
Computer Science Dept.

2331

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



A Constructive Approach to the Design of Algorithms and their Data Structures

Gaston H. Gonnet
Frank Wm. Tompa

CS-80-47

November, 1980

A CONSTRUCTIVE APPROACH TO THE DESIGN OF ALGORITHMS AND THEIR DATA STRUCTURES

Gaston H. Gonnet
Frank Wm. Tompa

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

ABSTRACT

The design of algorithms and data structures can usually benefit from a study of previous experience. A framework for describing specific algorithms and their data structures is introduced in order that designs can be presented in a uniform style that is suitable for discovering new designs as well as documenting known ones. Data objects are described in terms of a formal grammar and most data manipulation is characterized as a composition of a few simple algorithms. Descriptions for several standard algorithms are included to illustrate the process, and a few newly-designed structures are introduced. Such an approach is expected to be extremely useful in the construction of software libraries.

Key words and phrases: algorithms, algorithm classification, algorithm construction, data structures, data types, data representation, design, specification, W-grammar, functional composition.

CR *Categories:* 4.34, 4.9, 5.24, 5.29.

This work was supported by the Natural Sciences and Engineering Research Council of Canada under grants A3353 and A9292.

October 29, 1980

A CONSTRUCTIVE APPROACH TO THE DESIGN OF ALGORITHMS AND THEIR DATA STRUCTURES

Gaston H. Gonnet
Frank Wm. Tompa

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

1. INTRODUCTION

Much of recent research into data structures has been directed at representation-independent descriptions of (abstract) data types [Dungan79] or at descriptions of isolated implementations (e.g., [Standish80]). Unfortunately there has not been a concerted effort to describe implementations using a common framework. As a result, except in a few instances (e.g., [Darlington78, Batory80]), it has been difficult to appreciate the similarities among distinct representations and algorithms and to obtain insight into the design of modified representations to suit particular processing requirements.

In this paper, we present a framework for describing data structure implementations. The goal is to be able to specify both the (static) representational structures themselves and the algorithms that are used to manipulate them. We do not claim to possess a universal specification language, much less a unique nor an optimal presentation method. Rather we wish to present one specification framework that serves its intended purpose of

. describing a wide range of implementations in enough detail to allow an average programmer to code each effectively in some language,
. serving as a framework for describing data structures and algorithms, especially for educational purposes,
. encouraging the discovery of similarities among algorithms,
. encouraging the discovery of new implementations through modification of those already described.

Before proceeding with the methodology, we would like to contrast our approach to that of researchers in abstract data types. The goal of abstract data type specification is to describe objects in a representation-independent way in order that implementation detail can be suppressed; the result is a description that can be shown to be correct for some intended purpose, but that is still far-removed from programs. In contrast, our concentration on an operational description of algorithms allows programmers to compare and contrast implementations based on some measures of performance. For example, the descriptions will provide a contrast among various possible representations for dictionaries, rather than concentrating on the abstract properties of dictionaries themselves.

2. STATIC OBJECT DEFINITIONS

The formal description of data structure implementations is similar to the formal description of programming languages. In defining a programming language, one typically presents first a syntax for valid programs in the form of a grammar and then further validity restrictions (e.g., usage rules for symbolic names) in terms of constraints that are not captured by the grammar. Similarly, a valid instance of a data structure implementation will be one that satisfies a syntactic grammar and also obeys certain constraints. For example, for a particular data structure instance to be a valid weight-balanced binary tree [Nievergelt74], it must first satisfy the grammatical rules for binary trees, and it must also satisfy a specific balancing constraint.

2.1. Grammar for Data Objects

The syntax of a data object class can be defined using a W-grammar (also called a two-level or van Wijngaarden grammar) [van Wijngaarden65].† Actually the full capabilities of W-grammars will not be utilized; rather the syntax will be defined using the equivalent of standard BNF productions together with the uniform replacement rule as described below.

A W-grammar generates a language in two steps (levels). In the first step, a collection of generalized rules are used to create more specific production rules. In the second step, the production rules generated in the first step are used to define the actual data structures.

This two-step process can be illustrated as follows. A sequence of real numbers can be defined by the BNF production

<S> ::= [real , <S>] | nil .

Thus a sequence of reals can have the form nil, [real,nil], [real,[real,nil]], and so on. Similarly, sequences of integers, characters, strings, boolean constants, etc. could be defined. However, this would result in a bulky collection of production rules which are all very much alike. One might first try to eliminate this repetitiveness by defining

<S> ::= [<D> , <S>] | nil

where <D> is given as the list of data types

<D> ::= real | int | bool | string | char .

However, this pair of productions generates unwanted sequences such as

[real,[int,nil]]

as well as the homogeneous sequences desired.

In a W-grammar, the problem of listing repetitive production rules is solved by starting out with generalized rule-forms, known as *hyperrules*, rather than the rules themselves. The generalized form of a sequence S is given by the hyperrule

s-D : [D , s-D] ; nil .‡

† W-grammars are only one of several formalisms that could be used to specify the syntax. Equally powerful would be the syntax section of algebraic specifications languages (e.g. [Tompa80]) or parametric extensions to graph grammars.

‡ The metasymbol ; indicates alternation.

The set of possible substitutions for **D** are now defined in a *metaproduction*, as distinguished from a conventional BNF-type production. For example, if **D** is given as

D :: real; int; bool; string; char; ... .

a sequence of real numbers is defined in two steps as follows. The first step consists of choosing a value to substitute for **D** from the list of possibilities given by the appropriate metaproduction: in this instance, **D → real.** Next invoke the uniform replacement rule to substitute the string **real** for **D** everywhere it appears in the hyperrule that defines **s−D**. This substitution gives

s-real : [real , s-real] ; nil .

Thus the joint use of the metaproduction and the hyperrule generates an ordinary BNF-like production defining real sequences. The same two statements can generate a production rule for sequences of any other valid data type (integer, character, etc.).

Figure 1 contains a W-grammar which will generate many conventional data objects. As further examples of the use of this grammar, consider what happens when **D→real** and **LEAF→nil**. With these substitutions, HR[3] generates the production rule

bt−real−nil : [real, bt−real−nil, bt−real−nil] ; nil

which defines a binary tree that has a real entry in each node.† Since **bt−real−nil** is one of the legitimate values for **D** according to M[1] let **D→bt−real−nil** from which HR[1] indicates that such a binary tree is a legitimate data structure. Secondly consider a production rule for B-trees of strings using HR[4] and the appropriate metaproductions to yield

mt−string−nil : [**int**, $\{\textbf{string}\}_1^{10}$, $\{\textbf{mt−string−nil}\}_0^{10}$] : **nil.**

This is a multi-way tree in which each node contains ten keys and has eleven descendants.

Finally consider the specification for a hash table to be used with direct chaining. The production

s-(string,int) : [(string,int) , s-(string,int)] ; nil

and M[1] yield

D → $\{\textbf{s-(string,int)}\}_0^{96}$

Thus HR[1] will yield a production for an array of sequences of string/integer pairs usable, for example, to record NAME/AGE entries using hashing.

2.2. Constraints for Data Objects

Certain syntactic characteristics of data objects are difficult or cumbersome to define using formal grammars. For example weight balance for trees or lexicographic ordering for sequences can be defined using W-grammars, but they are more easily understood using constraints, similar in nature to a program's "*static semantics:* admonitions, restrictions, and other information about the form of the program" [Koster76].

† Throughout this paper, an acquaintance with standard data structures is assumed (see, for example, [Gotlieb78, Standish80]).

Metaproductions:

M[1]	**D** ::	**real;int;bool;string;char;...;**	# atomic data types #
		$\{ \mathbf{D} \}_{\mathbf{N}}^{\mathbf{N}}$;	# array #
		DS ; **(DS)** ;	# record #
		[D] ;	# reference #
		s−D;	# sequence #
		gt−D−LEAF;	# general tree #
		SEARCH;	# dictionary structures #
		... .	# other structure classes #
M[2]	**SEARCH** ::	$\{\mathbf{D}\}_{\mathbf{N}}^{\mathbf{N}}$; **s−D**;	
		bt−D−LEAF;	# binary tree #
		mt−D−LEAF;	# multiway tree #
		tr−D.	# trie #
M[3]	**DS** ::	**D**; **D, DS**.	
M[4]	**LEAF** ::	**nil**; **D**.	
M[5]	**N** ::	**DIGIT**; **DIGIT N**.	
M[6]	**DIGIT** ::	0;1;2;3;4;5;6;7;8;9.	

Hyperrules:

HR[1]	**data structure** :	**D**.
HR[2]	**s−D** :	**[D , s−D]** ; **nil**.
HR[3]	**bt−D−LEAF** :	**[D , bt−D−LEAF , bt−D−LEAF]** ; **LEAF**.
HR[4]	**mt−D−LEAF** :	[**int** , $\{\mathbf{D}\}_{1}^{\mathbf{N}}$, $\{\mathbf{mt{-}D{-}LEAF}\}_{0}^{\mathbf{N}}$] ; **LEAF**.
HR[5]	**gt−D−LEAF** :	**[D , s−gt−D−LEAF]** ; **LEAF**.
HR[6]	**tr−D**:	[$\{ \mathbf{tr{-}D} \}_{1}^{\mathbf{N}}$] ; **[D]** ; **nil**.

Figure 1: Grammar for data objects

A semantic rule or constraint may be regarded as a boolean function on data objects ($S: D \rightarrow$bool) that indicates which are valid and which are not. Objects that are valid instances of a data structure implementation are those in the intersection of the set produced by the W-grammars and those that satisfy the constraints.

Below are some examples of semantic rules which may be imposed on data structures. As phrased, these constraints are placed on data structures that have been legitimately produced by the rules given in the previous section.

Sequential order

Many data structures are kept in some fixed order (e.g. the records in a file are often arranged alphabetically or numerically according to some key). Whatever work is done on such a file should not disrupt this order.

Lexicographical trees

A lexicographical tree is a tree that satisfies the following condition for every node s: If s has n keys ($key_1, key_2, \ldots, key_n$) stored in it, s must have $n+1$ descendant subtrees $t_0, t_1, \ldots, t_n$. Furthermore, if d_0 is any key in any node of t_0, d_1 any key in any node of t_1, and so on, the inequality $d_0 \leqslant key_1 \leqslant d_1 \leqslant \ldots \leqslant key_n \leqslant d_n$ must hold.

Priority queues

A priority queue can be any kind of recursive structure in which an order relation has been established between each node and its descendants. One example of such an order relation would be to require that $key_p \leqslant key_d$, where key_p is any key in a parent node, and key_d is any key in any descendant of that node.

Height balance

Let s be any node of a tree (binary or multiway). Define $h(s)$ as the height of the subtree rooted in s, i.e. the maximum number of nodes one must pass through to reach the end of branch starting at s. One structural quality that may be required is that the height of a tree along any pair of adjacent branches be approximately the same. More formally, the height balance constraint is $|h(s_1)-h(s_2)| \leqslant \delta$ where s_1 and s_2 are any two subtrees of any node in the tree, and δ is a constant giving the maximum allowable height difference. In B-trees, for example, $\delta=0$, while in AVL-trees, $\delta=1$.

Weight balance

For any tree, the weight function $w(s)$ is defined as the number of external nodes (leaves) in the subtree rooted at s. A weight balance condition requires that for any two nodes s_1 and s_2, if they are both subtrees of any other node in the tree, $r \leqslant w(s_1)/w(s_2) \leqslant 1/r$ where r is a positive constant less than 1.

Optimality

Any condition on a data structure which minimizes a complexity measure (such as the expected number of accesses or the maximum number of comparisons) is an optimality condition. If this minimized measure of complexity is based on a worst case value, the value is called the *minimax*; when the minimized complexity measure is based on an average value, it is the *minave*.

In summary the W-grammars are used to define the general shape or pattern of the data objects. Secondly, once an object is generated, its validity is checked against the semantic rules or constraints that may apply to it.

3. ALGORITHM DESCRIPTIONS

Having defined the objects used to maintain data, it is appropriate to describe the algorithms that access them. Furthermore, because data objects are not static, it is equally important to describe data structure manipulation algorithms.

An algorithm is a function that operates on data structures. More formally, an algorithm is a map $S \rightarrow R$ or $S \times P \rightarrow R$, where S, P, and R are all structures; S is called the input structure, P contains parameters (e.g., to specify a query), and R is the result.† The two following examples illustrate these concepts:

(1) Quicksort is an algorithm that takes an array and sorts it. Since there are no parameters,

Quicksort: array → sorted-array.

(2) B-tree insertion is an algorithm that inserts a new record P into a B-tree S, giving a new B-tree as a result. In function notation,

B-tree-insertion: B-tree × new-record → B-tree.

Algorithms compute functions over data structures. As always, different algorithms may compute the same functions; $\sin(2x)$ and $2\sin(x)\cos(x)$ are two expressions that compute the same function. Since equivalent algorithms have different computational requirements, however, it is not merely the function computed by the algorithm that is of interest, but also the algorithm itself.

Following some of the ideas of structured programming, it is clear that algorithms are typically not indivisible units created in isolation, but rather they are built up from basics. However, one cannot define a set of basic operations for algorithms without paying some attention to the building procedures as well. After all, the richer the set of building procedures, the simpler (and possible fewer) atomic operations are needed to construct usable algorithms. On the other hand, with a large collection of basic operations, fewer building procedures may be needed to be able to construct the algorithms desired. There are an infinite number of ways to define basic operations and building procedures that will produce equivalent algorithms. We do not claim that the division of operations we make is optimal; our motivation for choosing the following system is simply that it seems natural. In the following section, we describe a few basic operations informally in order to convey their flavour.

3.1. Basic (or Atomic) Operations

A primary class of basic operations manipulate atomic values and are used to focus an algorithm's execution on the appropriate part(s) of a composite data object. The most common of these are as follows:

Ranking: set of scalars × scalar → integer

This operation is defined on a set of scalars $X_1, X_2, \ldots, X_n$ and uses as parameter another scalar X. Ranking determines how many of the X_j values are less than or equal to X, thus determining what rank X would have if all the values were ordered. More precisely, ranking is finding an integer i such that there is a subset $A \subseteq \{X_1, X_2, \ldots, X_n\}$ for which $|A| = i$ and $X_j \in A$ if and only if $X_j \leqslant X$. Ranking is used primarily in directing multiway decisions. For example, in a binary decision, $n=1$, and i is zero if $X < X_1$, one otherwise.

† With more than one input to an algorithm, the choice of which belong to S and which to P is somewhat arbitrary, typically made to emphasize one input over the others.

Hashing: value × range → integer

Hashing is an operation which normally makes use of a record key. Rather than using the actual key value, however, an algorithm invokes hashing to transform the key into an integer in a prescribed range by means of a hashing function and then uses the generated integer value.

Interpolation: numeric-value × parameters → integer

Similarly to hashing, this operation is typically used on record keys. Interpolation computes an integer value, based on the input value, the desired range, the values of the smallest and largest of a set of values, and the distribution of the values in the set. Interpolation normally gives the statistical mode of the location of a desired record in a random ordered file, i.e. the most probable location of the record.

Digitization: scalar → sequence of scalars

This operation tranforms a scalar into a sequence of scalars. Numbering systems that allow the represention of integers as sequences of digits and strings as sequences of characters provide natural methods of digitization.

Testing for equality: value × value → boolean

Rather than relying on multiway decisions to test two values for equality, a distinct operation is included in the basic set. Given two values of the same type (e.g., two integers, two characters, two strings), determine whether they are equal. Notice that the use of multiway branching plus equality testing closely matches the behaviour of most processors and programming languages, which require two tests for a three-way branch determining less than, equal, or greater than.

Other classes of basic operations relate to each of the data structure constructors used in the grammar for data objects. Corresponding to the array constructor $\{D\}_N^N$ is an access operation (commonly referred to as "indexing"), used either to read or to write a component's value; for a record, there is an operation to select a component field; and for a reference, there is an operation to access a referent.

3.2. Building Procedures

Building procedures are used to combine basic operations and simple algorithms to produce more complicated ones. In this section, we will define four building procedures.

3.2.1. Composition

Composition is the main procedure for producing algorithms from atomic operations. Typically, but not exclusively, the composition of $F_1{:}S \times P \rightarrow R$ and $F_2{:}S \times P \rightarrow R$ can be expressed in a functional notation as $F_2(F_1(S,P_1),P_2)$. A more general and hierarchical description of composition is that the description of F_2 uses F_1 instead of a basic operation.

Although this definition is enough to include all types of composition, there are several common forms of composition that deserve to be identified explicitly.

Divide and conquer uses a composition involving two algorithms for problem that are greater than a critical size. The first algorithm splits a problem into (usually two) smaller problems. The composed algorithm is then recursively applied to each non-empty component using recursion termination when appropriate. Finally the second algorithm is used to assemble the components' results into one result. A typical example of divide and conquer is quicksort (where the termination alternative may use a linear insertion sort). Diagramatically:

```
solve-problem(A):
    if size(A) ≤ critical
        then  {end action}
        else  {split problem}
              solve-problem(A1);
              solve-problem(A2);
               . . . .
              {assemble results}  fi
```

Iterative application operates on an algorithm and a sequence of data structures. The algorithm is iteratively applied using successive elements of the sequence in the place of the single element for which it was written. For example, insertion sort iteratively inserts an element into a sorted sequence.

```
solve-problem(S):
    while S is not exhausted do
        {apply algorithm to next element of sequence S};
        {advance S}  od;
    {end action}
```

or alternatively, if the sequence is in an array:

```
solve-problem(A):
    for i from 1 to |A| do
        {action on A[i]}  od;
    {end action}
```

Tail recursion is a composition involving one algorithm that specifies the criterion for splitting a problem into (typically two) components and selecting one of them to be solved recursively. A classical example is binary search.

```
solve-problem(A):
        if size(A) ≤ critical
                then   {end action}
                else   {split and select subproblem i};
                       solve-problem(A_i)  fi
```

or alternatively, unwinding the recursion into a while loop:

```
solve-problem(A):
        while size(A)>critical do
                {split and select subproblem i};
                {replace A by A_i} od;
        {end action}
```

It should be noted that tail recursion can be viewed as a variant of divide and conquer.

Inversion is the composition of two search algorithms used to search for sets of records based on values of secondary keys. The first algorithm is used to search for the selected attribute (find the "inverted list" for "hair colour" as opposed to "salary range") and the second algorithm is used to search for the set with the corresponding key value (e.g. "blonde" as opposed to "brown"). In general, inversion returns a set of records which may be further processed (for example, using intersection, union, or set difference).

```
inverted-search(S,A,V):
        # S is a structure, A an attribute, and V a value #
        search ( search(S,A), V )
```

Digital decomposition is applied to a problem of size n by attacking preferred-size pieces (for example, pieces of size equal to a power of two). An algorithm is applied to all these pieces to produce the desired result. One typical example is binary decompostion [Bentley79].

```
Solve-problem(A,n):
        # n has a digital decomposition: n=n_k β_k +...+n_1 β_1 + n_0 #
        {Partition the problem into subsets
                A = ∪_{i=0}^{k} ∪_{j=1}^{n_i} A_i^j }
                        # where |A_i^j| = β_i # ;
        for i from 0 to k while not completed do
                simpler-solve({A_i^1, A_i^2, ..., A_i^{n_i} })  od od
```

Merge applies an algorithm and a discarding rule to two or more sequences of data structures. The sequences are ordered on a common key. The algorithm is iteratively applied using successive elements of the sequences in place of the single elements for which it was written. The discarding rule controls the iteration process. For example, set union, intersection, merge sort, and almost all business applications use merging.

```
Merge(S_1,S_2,...,S_k):
        while (at least one S is not empty) do
                kmin := minimum value of keys in (heads of) S_1...S_k;
                for i from 1 to k do
                        if kmin = first key in S_i
                                then t_i := head of S_i
                                else t_i := null  fi;
                processing-rule( (t_1, t_2,..., t_k) )   od;
        {end action}
```

3.2.2. Alternation

The simplest form of building operation is alternation. Depending on the result of a test or on the value of a discriminator, one of several alternative algorithms is invoked. For example, based on the value of a command token in a batch updating interpreter, an insertion, modification, or deletion algorithm could be invoked; based on the success of a search in a table, the result could be processed or an error handler called; based on the size of the input set, an $O(N^2)$ or an $O(N \log N)$ sorting algorithm could be chosen.

There are several forms of alternation that appear in many algorithms; these are elaborated here.

Superimposition combines two or more algorithms, allowing them to operate on the same data structure more or less independently. Two algorithms F_1 and F_2 may be superimposed over a structure S if $F_1(S,Q_1)$ and $F_2(S,Q_2)$ can both operate together. A typical example of this situation is a file that can be searched by one attribute using F_1 and by another attribute using F_2. Unlike other forms of alternation, the alternative to be used cannot be determined from the state of the structure itself; rather superimposition implies the capability of using any alternative on any instance of the structure involved. Diagramatically:

solve-problem(A):
 case 1: solve-problem$_1$(A);
 case 2: solve-problem$_2$(A);
 ...
 case n: solve-problem$_n$(A)

Interleaving is a special case of alternation in which one algorithm does not need to wait for other algorithms to terminate before starting its execution. For example one algorithm might add records to a file while a second algorithm makes deletions; interleaving the two would give an algorithm that performs additions and deletions in a single pass through the file.

Recursion termination is an alternation that separates the majority of the manipulation for a structure from the end actions. For example, checking for end of file on input, for reaching a leaf in a search tree, or for reduction to a trivial sublist is a binary search are applications of recursion termination. It is important to realize that this form of alternation is equally applicable to iterative processes as to recursive ones. Several examples of recursion termination were presented in the previous section on composition (see, for example, divide and conquer).

3.2.3. Organization

If an algorithm creates or changes a data structures, it is sometimes necessary to perform more work to ensure that semantic rules and constraints on the data structure are not violated. For example, when nodes are inserted into or deleted from a tree, the tree's height balance may be altered. As a result it may become necessary to perform some action to restore the balance in the new tree. This process of combining an algorithm with a "clean-up" operation on the data structure involved is called *organization* (sometimes *reorganization*). In effect, organization is a composition of two algorithms: the original modification algorithm and the constraint satisfaction algorithm. Because this form of composition has an acknowledged meaning to the algorithm's users, it is convenient to list it as a separate class of building operation rather than as a variant of composition. Other examples of organization include reordering elements in a modified list to restore lexicographic order, percolating newly-inserted elements to their appropriate locations in a priority queue, and removing all dangling (formerly incident) edges from a graph after a vertex is deleted.

3.2.4. Self-organization

This is a supplementary heuristic activity that an algorithm may often perform in the course of querying a structure. Not only does the algorithm do its primary work, it also reaccomodates the data structure involved in a way designed to improve the performance of future queries. For example, a search algorithm may promote the searched element once it is found so that future searches through the file will locate this record more quickly. Similarly, a page management system may mark pages as they are accessed in order that "least recently used" pages may be identified for subsequent replacement.

Once again, this building procedure may be viewed as a special case of composition (or of interleaving); however, its intent is not to build a functionally different algorithm, but rather to augment an algorithm to include improved performance characteristics.

3.3. Interchangeability

The framework as described so far clearly satisfies two of its goals: sufficient detail to allow effective encoding in any programming language and uniformity of description to simplify teaching. It remains to be shown that the approach can be used to discover similarities among implementations as well as to design modifications that result in new useful algorithms.

The primary vehicle for satisfying these goals is the application of interchangeability. Having decomposed algorithms into basic operations used in simple combinations, one is quickly lead to replacing any component of an algorithm by something similar.

The simplest form of interchangeability is captured in the static objects' definition. The hyperrules emphasize similarities among the data structure implementations by indicating the universe of uniform substitutions that can be applied. For example, in any structure using a sequence of reals, the hyperrule for **s−D** together with that for **D** indicates that the sequence of reals can be replaced by a sequence of integers, a sequence of binary trees, etc. Algorithms that deal with such modified structures need at most superficial changes for manipulating the new sequences, although more extensive modifications may be necessary in parts that deal with the components of the sequence directly.

The next level of interchangeability results from the observation that some data structure implementations can be used to simulate the behaviour of others. For example, wherever a bounded sequence is used in an algorithm, it may be replaced by an array, relying on the sequentiality of the integers to access the array's components in order. Sequences of unbounded length may be replaced by sequences of arrays, a technique that is usefully applied to adapt an algorithm designed for a one-level store to operate in a two-level memory environment wherein each block will hold one array. This notion of interchangeability is the one usually promoted by researchers using abstract data types, who would claim that the algorithms should have been originally specified in terms of abstract sequences. We feel that the approach presented here does not contradict those claims, but rather that many algorithms already exist for specific representations and that an operational approach to specifying algorithms together with the notion of interchangeability is more likely to appeal to data structure practitioners. In

interpolation value *i*. This algorithm has the advantage that range searches are possible and efficient and that the file is very close to total order (only the sequences themselves are out of order).

- If hashing is composed with interpolation, in other words, interpolation is performed using the result of a hashing function applied on a key instead of using the keys themselves, a new algorithm results. The structure is again derived from $\mathbf{D} \rightarrow \{\mathbf{D}\}_{\mathbf{N}}^{\mathbf{N}}$, but the components are ordered by hash-function value rather than by key value. This algorithm is as efficient as interpolation search [Gonnet80a] and does not suffer the same difficulties as pure interpolation search does when the keys are not uniformly distributed.

- Composing interpolation with the linear collision resolution scheme produces an interesting algorithm which is similar to linear probing but which constructs an almost ordered table. From this a fairly efficient sorting method can be derived [Gonnet80b].

4. CONCLUSIONS

The descriptive framework presented in this paper simultaneously addresses the presentation of algorithms and their data structures. The basic operations for an algorithm correspond very closely to the metaproductions used to define the data structure; for example, indexing in an array corresponds to the metaproduction $\mathbf{D} \rightarrow \{\mathbf{D}\}_{\mathbf{N}}^{\mathbf{N}}$. The alternation and composition building procedures often deal with the particular hyperrules used to form the structure; for example, recursion termination typically corresponds to the use of "**;**" in a hyperrule, and the use of iterative application corresponds to the processing of a sequence **s-D**. The two "semantic" building procedures, organization and self-organization, correspond to the less formalized aspects of data structure description captured in the constraints as explained in Section 2.2.

This framework has been used at the University of Waterloo for teaching about data structure and algorithms. The approach captures the notions of modularity that many educators have found to be desirable. The level of formalism is sufficient to capture intuitive ideas of interchangeability and to translate algorithms into operating programs. At the same time, the constructive aspect of the approach is less intimidating to students than other formalisms have been.

Finally, the primary value of this constructive approach may be in the insight gained when designing new algorithms. Based on the experiences gained so far with interpolation algorithms, we feel that the approach has already proven itself, and we look forward to further new developments.

Acknowledgements. The authors have benefited from interactions with several researchers and teachers, especially Jon Bentley, who helped to clarify some of the ideas regarding composition.

References

[Batory80] Batory, D.S. and Gotlieb, C.C. A unifying model of physical databases. Univ. of Toronto CSRG Tech. Rept. CSRG-109 (1980), 39 pp.

[Bentley79] Bentley, J.B. Decomposable seaching problems, *Information Processing Letters 8*,3 (June 1979), 244-251.

[Darlington78] Darlington, J. A synthesis of several sorting algorithms. *Acta Informatica 11*, 1 (1978), 1-30.

[Dungan79] Dungan, D.M. Bibliography on data types. *SIGPLAN Notices 14*, 11 (Nov. 1979), 31-53.

[Gonnet77] Gonnet, G.H. and Rogers, L.D. The interpolation-sequential search algorithm. *Information Processing Letters 6*, 4 (Aug. 1977), 136-139.

[Gonnet80a] Gonnet, G.H., Rogers, L.D., and George, J.A. An algorithmic and complexity analysis of interpolation search. *Acta Informatica 13*,1 (Jan. 1980), 39-46.

[Gonnet80b] Gonnet, G.H. and Munro, J.I. Linear probing sort. (in preparation).

[Gotlieb78] Gotlieb, C.C. and Gotlieb, L.R. *Data Types and Structures*. Prentice-Hall, Englewood Cliffs, N.J., 1978, 444 pp.

[Koster76] Koster, C.H.A. Two-level grammars. *Compiler Construction: an Advanced Course*, 2nd edition (F. L. Bauer and J. Eickel, ed.) *Lecture Notes in Computer Science 21*. Springer-Verlag, 1976, 146-156.

[Nievergelt74] Nievergelt, J. Binary search trees and file organization. *Computing Surveys 6*, 3 (1974), 195-207.

[Standish80] Standish, T.A. *Data Structure Techniques*. Addison-Wesley, Reading, Mass., 1980, 447 pp.

[Tompa80] Tompa, F.W. A practical example of the specification of abstract data types. *Acta Informatica 13*, 2 (1980), 205-224.

[van Wijngaarden65] van Wijngaarden, A. Orthogonal design and description of a formal language. Math. Centrum Rept. MR76, Amsterdam, 1965.