STORAGE STRUCTURES FORMALISM

Lawrence A. Rowe and Fred H. Tonge

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92664

TECHNICAL REPORT #64 - April 1975

ABSTRACT

A storage structures formalism is described which can be used as a precise symbolic representation of low level storage organizations and also as a description of storage requirements to a storage allocation mechanism. The formalism is based on three forms of memory management (sequential, linked, and associative) and their associated referencing mechanisms (indexing, pointing, and hashing). Several examples of the formalism are presented. The meaning, or interpretation, of a storage structure as used to implement a modelling structure is discussed. Examples of alternative implementation structures (storage structures and their interpretations) for a particular modelling structure are presented.

INTRODUCTION AND OVERVIEW

Our view of the programming process is that problem domain information and algorithms are mapped into implementation domain representations by way of various modelling domain representations. One way to improve this process is to provide substantial machine assistance in mapping from modelling domain representations to implementation domain representations [TOW75]. In a previous paper [ROW74] we described a modelling structures formalism for representing problem domain information within a modelling domain. Representation of information in a modelling domain serves three purposes: (1) it provides a way for formally describing problem domain information without commitments to underlying implementation representations, (2) in many cases, it eases the task of expressing and understanding a problem solution procedure, and (3) it makes easier the discovery of alternative implementation representations.

We are developing a system which, given a modelling structure representation of the information and a description of its use in a particular problem solution procedure, assists in the generation, evaluation, and selection among alternative implementation representations. To do this we must be able to describe different

implementation representations, both the data and procedures for manipulating the data, and we must be able to incorporate in a useable database specific knowledge concerning possible implementation representations for modelling representations.

There are three levels of implementation domain knowledge to be considered. The lowest level includes the notions of indivisible cells (capable of representing primitive values) and of groupings of cells into structures. Such structures may be implemented using contiguous groupings of cells (referenced by a conventional indexing mechanism), cells which point explicitly to other cells or groups of cells (referenced by a conventional linking mechanism), and cells whose value serves as an argument to a structure-defining function (referenced by a hashing mechanism). These low level representations, which we call storage structures, do not include the procedures for using them, nor do they define the correspondence of their components to the modelling structures that they represent.

At an intermediate level is knowledge we call implementation strategies. These strategies are concerned with storage structures and their relationship with the modelling structures they are being used to implement. We refer to the result of applying this implementation

knowledge to generate a storage structure as an implementation structure. This implementation structure includes the storage structure, procedures for its use, and correspondences with the modelling structure it represents. We call the latter two items an interpretation of the storage structure. A particular type of storage structure can have many different interpretations.

The highest level of implementation knowledge concerns how several storage structures coexist in a block of memory. This includes knowledge about memory allocation and deallocation.

The second section of this paper describes a storage structures formalism. The motivation for the development of this formalism derives from its intended uses: (1) as a precise symbolic representation of low level storage organizations, (2) for representing the storage usage aspect of mappings from modelling structures to alternative implementation structures, and (3) as a description (request for storage) to be given to a storage allocation mechanism. Because this formalism is concerned with the lowest level of the implementation representation, a storage structure as represented in the formalism does not capture all the information of a modelling structure. Only at the level of the storage structure interpretation is all information in a

modelling structure represented.

Following the description of the formalism, a third section presents examples of storage structures. These include most of the "storage structures" provided in other implementation representation synthesis and selection systems [GOT74, LOW74]. The fourth section discusses interpretations of storage structures. The final section summarizes what we have presented.

STORAGE STRUCTURES FORMALISM

Description of the Formalism

The primitive indivisible entities of the formalism are cells capable of representing values. Each value has a type which names the class to which the value belongs. New entities, or types, can be created by composing entities using the assumed memory management forms: sequential, linked, or associative. Given two entities, named e and f, there are three composition operators:

- e+f e and f appear contiguously in memory,
- e@f e is linked to f, and
- e?f there exists a structure-defining function on e (based either on e's content or address, not specified by the formalism) which yields a reference to f.

Multiple instances of an entity can be expressed by

- n(e) n distinct instances of an e, and
- #(e) an indeterminate number of instances of e.

A composition operator can be distributed over a collection of entity instances by an APL-like distributive rule

$$e\ op\ e\ op\ .\ .\ .\ op\ e = n(e)/op.$$

The possible occurrence of alternative entities is

given by

e!f either an e or an f.

The linking operator (@) also serves as a unary operator denoting a pointer to an entity. For example,

@e

means a pointer or link to an e. Thus, storage for a link to an e is declared.

Entities can be named, for example

e : n(f)/+

names an entity e which is composed of n contiguous instances of f. For convenience, particular entities or operators can be indexed by subscripts. (Subscripting is necessary in expressing the interpretation of a storage structure.)

A storage structure made up of several different constructs without explicit description of their interconnection is represented by joining the constituent constructs with a semicolon (;). For example,

#(e);#(f)

is a structure composed of an arbitrary number of e's and an

arbitrary number of f's.

Four simple examples are:

n(e)/+	n instances of e stored contiguously.
#(e)/@	An arbitrary number of e's linked together.
n(e!f)/+	An indeterminate number each of e's or f's stored contiguously. The total number of entities is n.
cell@e e:(f+2(@e)/+)	An empty cell linked to an e. An e is composed of an instance of an f and two links to e's stored contiguously. Thus, there is an arbitrary number of e's. (Notice that this storage structure does not describe how the e's are connected, merely that each e includes space for a pointer to an e.)

An Interpretation of the Composition Operators

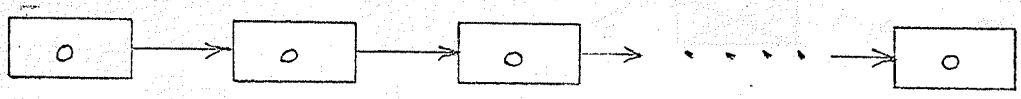
The three composition operators can be viewed as corresponding to three different storage management schemes: sequential storage, linked storage, and associative storage. Thus, if the storage allocation system within which the storage structures are to reside provides linked blocks of storage "automatically" (as, for example, in L* or the IPL series of languages), the composition operator @ could be used directly to link entities. On the other hand, if the storage management scheme as visible to the user consists of blocks of sequential storage available independently on request, the storage structure definition must include space for user-provided pointers from block to block, using the

unary operator @. In most contemporary architectures, only sequential storage is available at the machine and assembly language level.

EXAMPLES OF STORAGE STRUCTURES

The example storage structures specified in this section are taken from the implementation representation selection systems of Gotlieb and Tompa [GOT74] and Low [LOW74]. Each example includes a graphic representation for the storage structure, a description of what it is, and its specification in our formalism. In the examples, "o" corresponds to the actual object information.

Example 1: 1-way linked list

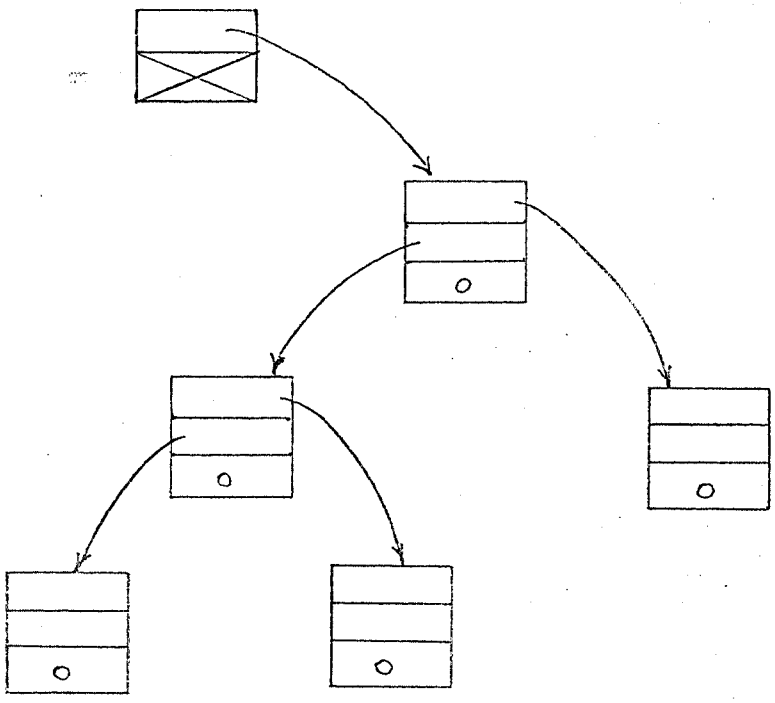


#(o)/@

Example 2: 2-way linked list

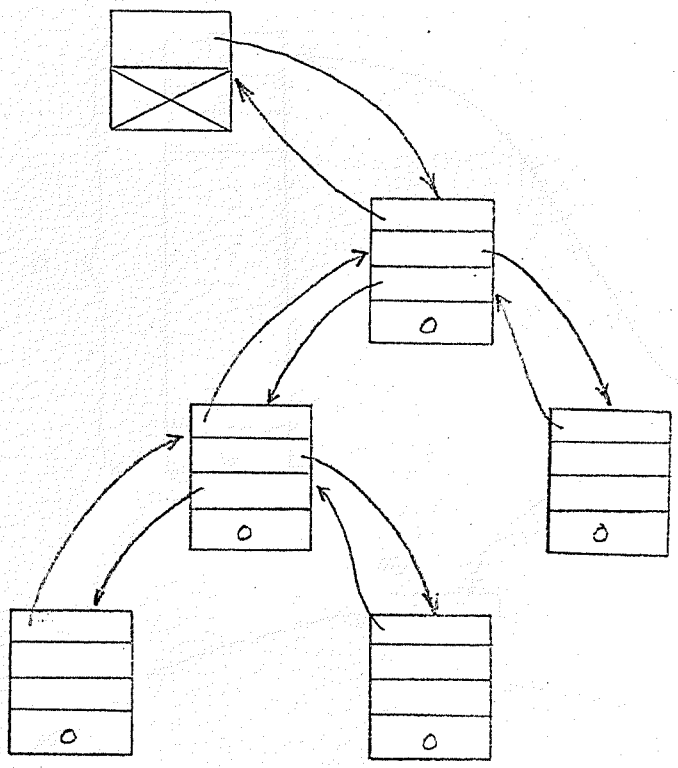
```
#(e)
e:o+2(@e)/+
```

Example 3: Binary Tree



```
cell@e; n(e)
e:o+2(@e)/+
```

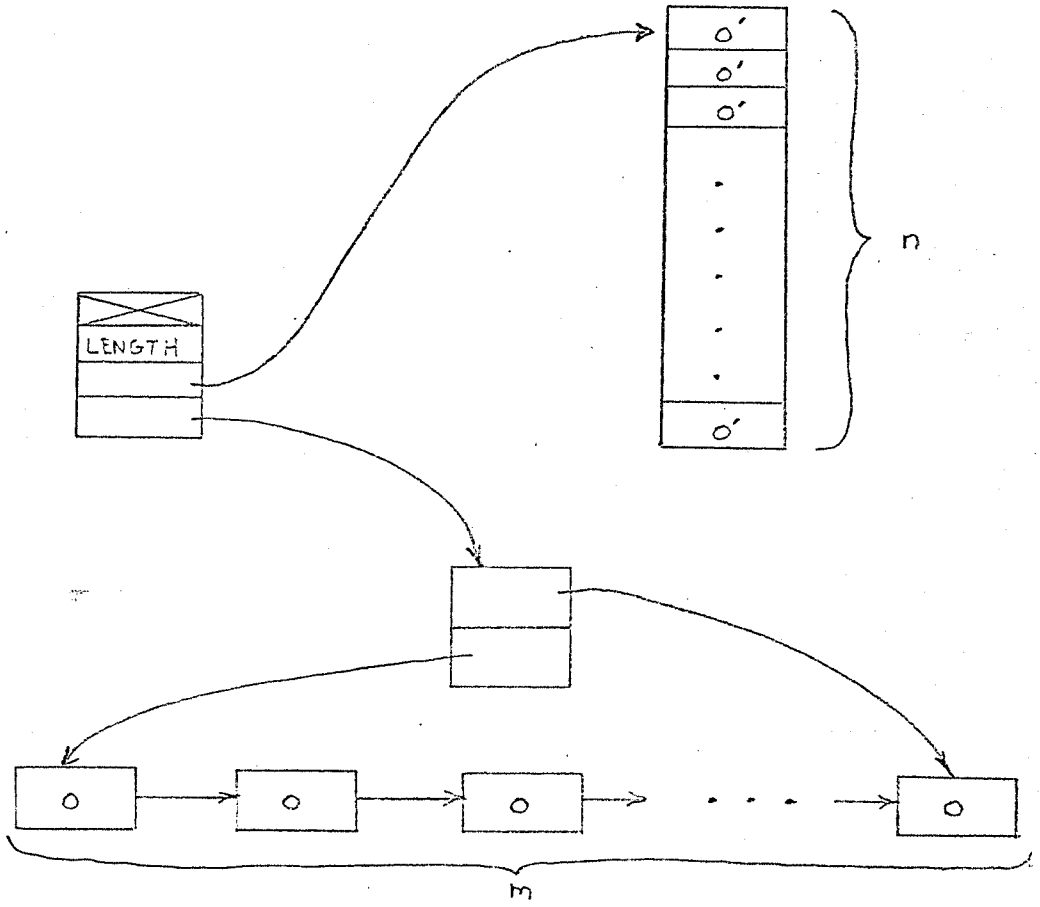
Example 4: Fathered Binary Tree



cell@e; n(e)

e:o+3(@e)/+

Example 5: Combination Fixed Length Bitstring and Linked List

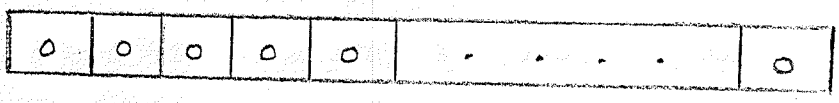


(2(cell)/+)+@o´+@f; n(o´)/+; m(e)

e:o+@e

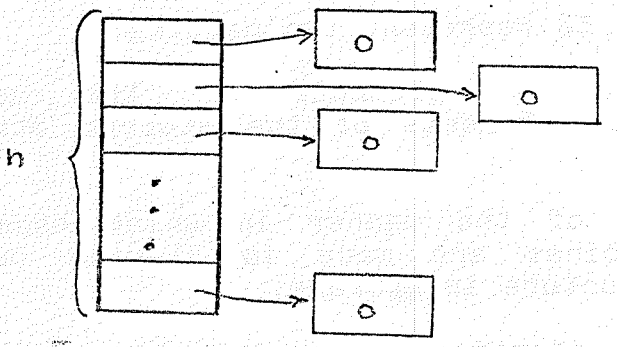
f:2(@e)/+

Example 6: Variable Length Array



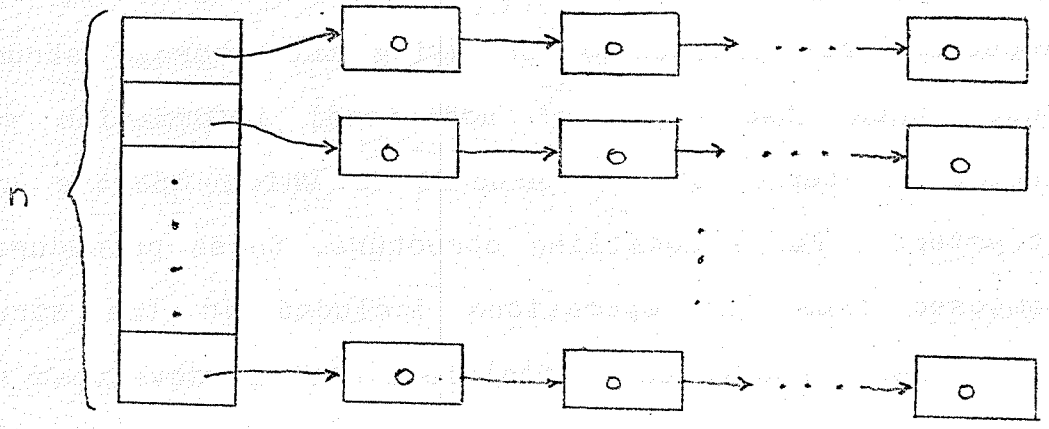
#(o)/+

Example 7: Contiguously Stored Links to Objects



n(@o)/+; n(o)

Example 8: Contiguously Stored Links to 1-way Linked Lists



n(@o)/+; n(#(o)/@)

INTERPRETATION OF STORAGE STRUCTURES

As defined above, a storage structure does not completely represent the modelling structure from which it is derived. That is to say, there is loss of information in the mapping process from modelling structures to storage structures. Two types of additional information in particular are needed to represent the mapping:

A) correspondence of elements of the modelling structure with elements of the storage structure, and

B) specification of the manner in which essentially arbitrary choices are made in "laying out" the modelling structure in storage.

Information of type A identifies corresponding parts of the two structures. Information of type B relates to the storage structure (and modelling structure) as a whole, and typically will imply additional storage requirements necessary for processing or using the storage structure. Thus, these two types of additional information can be viewed in terms of the procedures which operate upon a structure. For a modelling structure, these procedures are composed from the operations included in the structure definition. Equivalent definitions must be developed at the storage structure level. These storage structure operations depend not only on the representation of each element in storage (type A information), but also on overall

considerations which, for example, preclude or permit the use of stacks in processing (type B information).

We call A and B together the interpretation of the storage structure. In this section we develop the notion of a storage structure interpretation informally through several examples. In a later paper we shall present a formal description of interpretation.

Preliminaries

Related to each type of storage management scheme are certain bounding or limiting conventions. In contiguous storage, this is expressed as block size or maximum dimension; in linked storage, as a null link; in associative storage, as some means of signifying the absence of a value and, if such are possible under the particular scheme, handling collisions. For this paper we assume that such conventions are known to the mapping process and that the necessary processing steps and tests are provided. We also assume the existance of conventions for null pointers (@e) and for determing the extent of unspecified numbers of elements (#). In the following examples we omit these considerations.

Modelling Structure -- Binary Tree

The following information defines a binary tree in the

modelling structures formalism [ROW74].

replication

yes

ordering

no

relations

left (1-1, partial domain, partial range, not connected)
right (1-1, partial domain, partial range, not connected)
ansc (many-1, unique domain, partial range, connected)
ansc = $left^{-1} \cup right^{-1}$

distinguished elements

root - ansc(root) is undefined

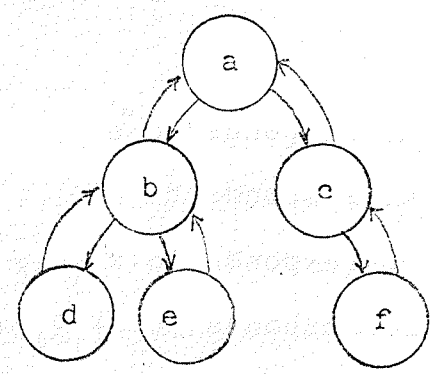
referencing

external access
distinguished element

operations

read
delete
replace
createaccess
relate
related

An example of this modelling structure might be represented pictorially as follows:



Next we give three example storage structures for this modelling structure, with their respective interpretations. In each case the interpretation must provide correspondences (type A information) for: the binary tree itself, the tree element, the relations left, right, and anse, and the distinguished element root.

Storage Structure I

a	$\vec{b}$	$\vec{c}$	b	$\vec{d}$	$\vec{e}$	c	$\vec{0}$	$\vec{f}$	d	$\vec{0}$	$\vec{0}$	e	$\vec{0}$	$\vec{0}$	f	$\vec{0}$	$\vec{0}$

where $\vec{x}$ represents a pointer to element x and $\vec{0}$ represents a null pointer.

Formally:

e : #(g)

g : $(f +_1 \ell_1 g +_2 \ell_2 g)$

f : <elemental data values>

(Henceforth, we use the shorthand notation $e:\#(f+_1\ell_1 e+_2\ell_2 e)$ where there is no danger of confusion.)

Interpretation:

the binary tree	corresponds to e
an element	corresponds to f
x left y	corresponds to $x+_1 @_1 y$
x right y	corresponds to $x+_1 @_1 z+_2 @_2 y$

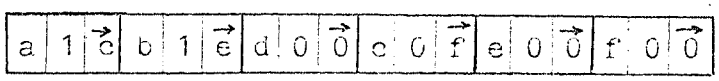
(Henceforth we use the shorthand notation left corresponds to $+_1 @_1$ and right corresponds to $+_2 @_2$ where there is no danger of confusion.)

x ansc y	corresponds to $(y+_1 @_1 x) \vee (y+_2 @_2 x)$
the root (is x)	corresponds to $\exists x \forall y \neg[(y+_1 @_1 x) \vee (y+_2 @_2 x)]$

(For the other examples we shall omit consideration of ansc and root, since they are derivable from left and right.)

Note that the relations left and right are bounded by the null pointer convention. Note also that no type B information appears necessary. (Arbitrary choices must be made in laying out individual elements in storage, but they do not affect processing routines.)

Storage Structure II



Formally:

e : $\#(f +_1 h +_2 @_2 e)/+_3$

f : <elemental data values>

h : 0 ! 1

Interpretation:

the binary tree corresponds to e

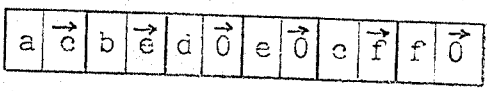
an element corresponds to f

left corresponds to $+_3 \wedge +_1 1$

right corresponds to $+_2 @_2$

Note that the relation <u>left</u> is bounded by $+_1 0$ and that <u>right</u> is bounded by the null pointer convention. Again, no type B information appears necessary.

Storage Structure III



Formally:

e : $\#(f +_1 @_1 e)/+_2$

f : <elemental data values>

Interpretation:

the binary tree corresponds to e

an element corresponds to f

x left y corresponds to $x +_2 y \wedge \forall z \neg [z +_1 @_1 y]$

right corresponds to $+_1 @_1$

Note that the relation <u>left</u> is bounded by the extent of the block (#) and the absence of a "right" pointer, and that

right is bounded by the null pointer convention. Note also that there are several alternative methods of "laying out" the elements in storage, and that these can affect the time and space required for processing. More specifically, the methods affect the handling of the quantified expression in the definition of the relation left, whose evaluation requires that potentially all other elements be examined. Thus, type B information is required for a complete interpretation.

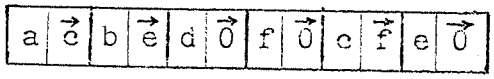
Three possible completions of the interpretation for storage structure III are given below. To consider these cases which deal with the arbitrary ordering of elements in contiguous storage, we introduce an index or address function i(x), defined as follows:

$i(x) < i(y) \iff$ x preceeds y in contiguous storage

Case 1:

No additional interpretation.

example:

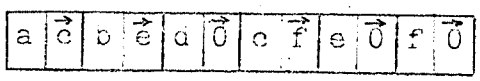


In this case, each element must be checked against all pointers in the structure.

Case 2:

$x +_1 @_1 y \iff i(x) < i(y)$

example:

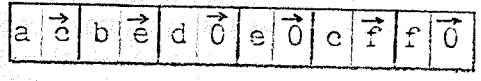


In this case an element must only be checked against pointers preceeding it in the structure.

Case 3:

$$x +_1 \in_1 y \iff i(x) < i(y) \wedge$$
$$\forall w [i(x) < i(w) < i(y) \wedge w +_1 \in_1 z \Rightarrow (i(z) < i(y))]$$

example:



In this case, an element must only be checked against the nearest preceeding pointer not matched by a preceeding element; e.g. against the top element of a stack of as yet unmatched pointers.

Remarks

The introduction of an index function allows a straightforward way of specifying the root as the "first" element in contiguous storage.

It would require a highly sophisticated mapping process to generate all of these alternatives. Their appearance here does not imply that we have on hand such a process; they are merely intended as examples of possible storage structures. (However, Schneiderman [SCH74] has claimed that the transformation from the equivalent of our storage structure I to storage structure III is straightforward.)

SUMMARY

The storage structures formalism presented here provides a precise symbolic representation of the storage requirements of an implementation structure. It permits specification of structures within sequential, linked, and associative storage management schemes, and allows fixed and arbitrary numbers of atomic data and pointers. To complete the description of the mapping from modelling structures to implementation structures, an interpretation of the storage structure is needed. An interpretation consists of two types of knowledge: correspondence of elements of the modelling structure with elements of the storage structure, and specification of arbitrary ordering choices in storage. While it is unlikely that an algorithm can be discovered which generates all possible implementation structures for a given modelling structure, this formalism and knowledge organization provides a framework for developing an acceptable one.

REFERENCES

[GOT74] Gotlieb, C. C. and F. W. Tompa. Choosing a Storage Schema. ACTA Informatica, vol. 3 (1974), pp. 297-319.

[LOW74] Low, J. R. Automatic Coding: Choice of Data Structures. Report CS-452, Computer Science Department, Stanford University (August 1974).

[ROW74] Rowe, L. A. Modelling Structures Formalism. Technical Report #52, Department of Information and Computer Science, U. C. Irvine (November 1974).

[SCH74] Schneiderman, B. Towards a Theory of Encoded Data Structures and Data Translation. Technical Report #13, University of Indiana (July 1974).

[TON75] Tonge, F. M. and L. A. Rowe. Data Representation and Synthesis. Technical Report #63, Department of Information and Computer Science, U. C. Irvine (March 1975).