Designing Data Structures

Alastair Reid

Abstract

The *design* (as opposed to the choice and use) of data structures has been the subject of relatively little study in the context of formal methods. In this paper, we introduce our ideas on how data structures are designed.

1 Introduction

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.
— John von Neumann

Implementation of a specification using data refinement is (roughly speaking) based on repeatedly choosing part of the specification and replacing it with a refinement which does "at least as much" and is (presumably) more implementable or efficient. Choosing a refinement may often be viewed as mere selection of a previously developed refinement of a similar specification from some form of library; but occasionally a new refinement must be designed — thus extending the library. Unfortunately, very little has been written about how these new data structures may be designed — [1], [2] and [3] being the most notable exceptions.

This paper is an introduction to the way we think about the design of efficient data structures. We shall be concerned mostly with time efficiency although we recognise that space efficiency is also important. We begin by briefly describing our notation and semantics. We then examine in some detail an example implementation of a specification paying particular attention to the data structure and attempting to draw general conclusions from our analysis. Finally, we discuss some of the problems with our approach.

2 Definitions

Miranda is used for our example specification. We give a brief overview of Miranda notation below. Further details may be found in [4].

- $[A]$ is the set of all lists $[a_1, \ldots a_m]$ with elements drawn from A and $m \in \mathbb{N}$.
- $f :: A \to B$ states that f is a (Miranda) function with *source type* A and *target type* B.
- $(++) :: [A] \to ([A] \to [A])$; $[a_1, \ldots a_m]++[a_{m+1}, \ldots a_n] = [a_1, \ldots a_m, a_{m+1}, \ldots a_n]$.
- $\texttt{length} :: [A] \to \texttt{num}$; $\texttt{length}[a_1, \ldots a_m] = m$.

- `head` $:: [A] \to A$; `head`$[a_1, \ldots a_m] = a_1$.
- `tail` $:: [A] \to [A]$; `tail`$[a_1, a_2, \ldots a_m] = [a_2, \ldots a_m]$.
- `init` $:: [A] \to [A]$; `init`$[a_1, \ldots a_{m-1}, a_m] = [a_1, \ldots a_{m-1}]$.

We may consider functions as sets of pairs in the usual set-theoretic way. For example, the functions which doubles every natural number is:

$$\text{double} = \{\langle 0,0\rangle, \langle 1,2\rangle, \langle 2,4\rangle, \langle 3,6\rangle, \ldots\}$$

For any function $f\colon A \to B$, $\mathit{dom}\, f = A$ and $\mathit{ran}\, f = B$.

We regard data structures as the representations of states. In order to apply our approach, we shall assume that the specifier has defined the following sets:

- A set X of *state names*.
- A set M of *modifiers*. Modifiers are names of total operations with arity $X \to X$. X is closed under application of the operations named in M.
- A set O of *observers*. Observers are names of total operations with domain X and range $\neq X$.

Informally, the basis for the specifier's choice of M and O is that modifiers are used to *change* the state and observers are used to *inspect* the state.

Our semantics (of specifications) is a special case of observational equivalence to the initial model of a specification.

Let R be a set of representations of the states named in X, *rep* a function mapping state names to representations, *imp* a function mapping operations (i.e. modifiers and observers) to their implementations. Then, we require that for all $m \in M$, $o \in O$ the diagrams in figures 1 and 2 commute.

$$\begin{tikzcd}
\mathrm{X} \arrow[r, "m"] \arrow[d, "rep"'] & \mathrm{X} \arrow[d, "rep"] \\
\mathrm{R} \arrow[r, "imp\, m"'] & \mathrm{R}
\end{tikzcd}$$

Figure 1

$$\begin{tikzcd}
\mathrm{X} \arrow[r, "o"] \arrow[d, "rep"'] & ran\, o \\
\mathrm{R} \arrow[ur, dashed, "imp\, o"'] &
\end{tikzcd}$$

Figure 2

That is,

$$\begin{aligned} Correct(\langle imp, rep\rangle) \overset{\text{def}}{=} \forall x : X, m : M, o : O \bullet\ & (imp\, m)(rep\, x) = rep(mx) \wedge \\ & (imp\, o)(rep\, x) = ox \end{aligned}$$

One of the consequences of this definition is that representations must be adequate — that is only states which cannot be distinguished using the available operations (i.e. observers and modifiers) may have the same representation as each other. This is an important property because it allows us to test a representation independently of the implementations of the operations.

Formalising the notion of states being indistinguishable using only a set of observers O and any sequence of modifiers from $[M]$, we define the equivalence $\equiv (\operatorname{mod} O)$:

$$x \equiv y \, (\operatorname{mod} O) \overset{\text{def}}{=} \forall n : \mathbb{N}, m_1 \ldots m_n : M, o\colon O \bullet om_1 \ldots m_n x = om_1 \ldots m_n y$$

Given a representation function *rep*, we say that *rep is adequate* iff *Adequate*(*rep*).

$$Adequate(rep) \quad \overset{\text{def}}{=} \quad \forall x, y : X \bullet rep\, x = rep\, y \Rightarrow x \equiv y \pmod{O}$$

We mention the special case that, if *rep* is an injection, *rep* is adequate.

Finally, we intend that specifications will be implemented in a number of stages and therefore we assume that the representation of the target type of the observers will be performed separately (if at all).

3 Analysis Of A Data Structure

In order to gain a better understanding of data structures, we examine a typical implementation of a double ended queue — gradually reversing the design process which (we believe) created it.

We begin by giving an example specification and an implementation of it; relating some of the definitions given in the previous section to the specification. We then consider data structures, starting with structural aspects and then looking at the values stored in a data structure. Finally we summarise this section with an outline of our approach.

3.1 A Specification And Its Implementation

The queue specification is given in figure 3. For ease of comprehension we use Miranda for our specification. Our use of a programming language for the specification might prompt the reader to think of our *specification* as an *implementation*. We point out that the distinction between the two is rather fuzzy and that, since our approach is based on the *semantics* of the specification rather than on more *syntactic* aspects, any distinction which may exist between specifications and implementations is avoided. As a consequence we mention that it is perfectly legitimate to consider our method as a (somewhat indirect) approach to *program transformation*.

```
queue == [char]
taggedChar ::= Tagc char | Error
eq    ::  queue                    || empty queue
front ::  queue -> taggedChar      || front of queue
add   ::  num -> queue -> queue    || add to rear
rem   ::  queue -> queue           || remove from rear
deq   ::  queue -> queue           || dequeue from front
eq = []
front [] = Error
front q  = Tagc head q
rem [] = []
rem q  = init q
deq [] = []
deq q  = tail q
add a q = q ++ [a]
```

Figure 3

Note that we have used the composite type `taggedChar` to make `front` total. In our discussion, we shall occasionally refer to the length of a queue, this is simply the length of the list representing the queue in the specification.

In this example, the only sensible choice of state names is the set of all terms of type `queue` and, because of their arity and usage, `rem` and `deq` are modifiers and `front` is an observer. Intuitively, `add` is also a modifier; we use partial application to produce the functions (`add‘a’`), (`add‘b’`), …which have the correct arity for modifiers.

The resulting sets are:

- $X = \{$`eq`, (`add‘a’`)`eq`, (`add‘b’`)`eq`, `deq`(`add‘a’`)`eq`, (`add‘c’`) `rem` `eq`, …$\}$
- $M = \{$`rem`, `deq`, (`add‘a’`), (`add‘b’`), …$\}$
- $O = \{$`front`$\}$

Our implementation (figure 4) is in a Pascalesque language and uses linked data structures. It is intended to be similar to implementations which would be derived by any competent programmer. Note that we have used the composite type `taggedChar` from the specification and that as we mentioned at the end of section 2, this type would have to be implemented (perhaps by a variant record) before the implementation could be used.

```
type    dq = record front, rear : ^cell end;
        cell =  record
                    item : num;
                    next, prev : ^cell
                end;
var     q:dq;

procedure EQ;
begin q.front := nil; q.rear := nil end;

function FRONT: taggedChar;
begin
  if q.front ≠ nil
    then FRONT := Tagc q.front^.item
    else FRONT := Error
end;

procedure ADD(x:char);
var t:^cell;
begin
  new(t); t^.item := x; t^.next:=nil; t^.prev := q.rear;
  if q.rear ≠ nil
    then q.rear^.next := t
    else q.front :=t;
  q.rear := t end
end;

procedure DEQ;
```

```
var t:^cell;
begin
  if q.front ≠ nil
    then begin
      t:=q.front; q.front := q.front^.next;
      if q.front ≠ nil
        then q.front^.prev := nil
        else q.rear:=nil;
      dispose(t)
  end
end;

procedure REM;
var t:^cell;
begin
  if q.rear ≠ nil
    then begin
      t:=q.rear; q.rear := q.rear^.prev;
      if q.rear ≠ nil
        then q.rear^.next:=nil
        else q.front:=nil;
      dispose(t)
  end
end;
```

Figure 4

We shall consider linked data structures as being made up of a number of *nodes* connected by unidirectional *links* and accessed from outside the structure by *entry points*. Figure 5 shows the representation used by the example implementation using the traditional style of data structure diagram [5, pp. 44–45].

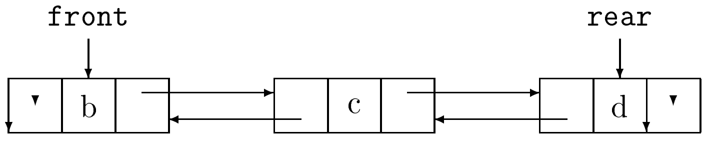
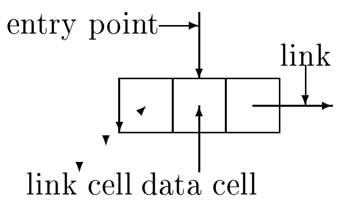


Figure 5



Key

Each node contains a number of labelled *link cells* (e.g. `prev` and `next`) and labelled *data cells* (e.g. `item`). We shall refer to the node pointed to by an entry point p as *the target of* p or as *the* p *node*. We shall refer to (the node containing) the link cell containing a link l as *the source (node/cell) of* l and the node pointed to by l as *the target node of* l. We allow the contents of a data cell to be structured using (variant) records, arrays, etc. and so, without loss of generality, we shall assume that each node contains a single data cell.

> We define a *path* in a data structure to be a list of link labels. If there is a sequence of nodes $[n_0, \ldots n_m]$ joined by a sequence of links $[l_1, \ldots l_m]$ such that the source and target of each link l_i are n_{i-1} and n_i respectively,

then we say there is *a path* $[p_1, \ldots p_m]$ *from* n_0 *to* n_m where p_i is the label of the link cell storing l_i.

For example, in figure 5, there are the following:

- a path [`next`, `next`] from the `front` node to the `rear` node;
- a path [`prev`, `prev`] from the `rear` node to the `front` node;
- a path [`next`, `prev`, `next`] from the `front` node to the central node;
- $\vdots$

In order to discuss efficiency, we require metrics. As a crude time metric, we shall count the number of primitive operations (traversal, removal, modification, etc. of entry points, links, values, etc.) and, as a space metric, the number of data cells, link cells and entry points in a representation.

3.2 Entry Points

Consider the use of entry points made by the implementations of observers and modifiers.

- Observers such as `FRONT` use entry points to determine which value to return. The position of observation points (i.e. entry points used by the observer) is therefore determined only by the choice of *representation of the current state*.
- Modifiers such as `ADD('e')` and `REM` use entry points to determine which change to make and then to make that change. Thus, the position of update points (i.e. entry points used to change the data structure) is determined by both *the representation of the current state* and *its relationship to the representations of the states derivable from it.*

Clearly the `rear` entry point is an update point; but the `front` entry point appears to be both an update point and an observation point. We believe that each entry point was introduced for a single specific purpose and that the `front` entry point is therefore the result of *fusing* two previously separate entry points — one an update point, the other an observation point.

We may generalise this idea of fusing two entry points with the same target to fusing two entry points with "nearby targets". We refine the term "nearby targets" with "approximation" which is defined as follows:

> Let b and c be entry points, p a path and R a class of representations of states. b p-approximates c in R iff in every representation $r \in R$, there is a path p from b to c.

For example, an entry point whose target is the penultimate node in a queue (i.e. the node adjacent to the `rear` node) is [`prev`]-approximated by the `rear` node (in all queues of length 2 or more). Such an entry point has indeed been "optimised out" — `REM` requires access to this node to allow the link to the old `rear` node to be removed.

We may generalise this notion still further by allowing the path defining the path from b to c to be a function of the representation. In this way, the `rear` entry point could be "optimised out" since it may be reached by following the `next` links from the `front` cell. Since the cost of this reduction in space is relatively high, this optimisation would probably not be justifiable purely in terms of the time-space tradeoff it represents. However, in examples like a priority queue [6, pp. 150–151] (or almost any other problem which is usually solved using some form of search) the cost of having an entry point at every cell where the implementation of a modifier may make a change is quite considerable e.g. every `ADD(x)` in a priority queue could require access to a different cell and so an individual entry point is required for every $x \in$ `char`. The only way to avoid this problem is to introduce loops using this generalisation of approximation (perhaps introducing additional structure to aid the location of the cell to be examined or modified). The problem of searching is a large one with many different solutions and so we shall not discuss it further here.

3.3 Links

We turn now to considering the use of links. With the exception of using links when one entry point approximates another, their sole purpose is to allow entry points to be moved when the implementation of a modifier is applied. This can be seen in the `prev` links whose only purpose (apart from the `[prev]`-approximation mentioned earlier) is to allow the `rear` update point to be moved when `REM` is applied.

From this it follows that after having:

1. chosen the parts of the representation of each state to be changed when the implementation of each modifier is applied; and

2. introduced entry points providing access to these points,

links are introduced between temporally adjacent targets of each entry point (where applying operations to the state corresponds to the flow of time). That is, if an entry point p points to a node b in the representation of some state x and p points to a node c in the representation of σx (for some $\sigma \in M$), then there should be a link from b to c in the representation of x (assuming c is in the representation of x).

Of course, since we have added new links to the representation, further changes will need to be made by the implementation of each modifier and so more entry points are introduced. If these cannot be fused with or approximated by the existing entry points, further links must be added requiring still more complex implementations of modifiers, more entry points, and so on ad infinitum. We thus see one of the other uses of fusion and approximation as being an *attempt* to avoid getting stuck in this loop.

Although it is not used in this example, there is a counterpart of entry point fusion for links. Clearly, if the links from two link cells b and c have the same target in all representations, they provide the same information and we may fuse the cells. Thus if there were links "running parallel to" the `next` links in the representation of the queue, we could fuse them with a substantial saving in space.

We may generalise this using *link approximation* in an analogous way to our generalisation of node fusion. We define link approximation as follows:

Let l be a link cell label, p a path and R a class of representations of states. p approximates l in R iff in every representation $r \in R$, for every l-link with source b and target c, there is a path p from b to c.

So far, we have discussed the "structural part" of the representation (i.e. links and entry points) and we have shown how the design of the structure is largely determined by:

- the *choice* of where to store the information required to make the observations; and
- the *choice* of where the modifiers access to change the representation.

We shall turn now to the "data part" of the representation where these choices are made.

3.4 Data Related Aspects Of Design

The task in designing the "data part" of an implementation is, essentially, deciding what features of the state to store in each representation so that the observers and the modifiers may be efficiently implemented. In this section, we look first at how we may represent the data part and then at the demands placed on the data part by correctness and efficiency requirements.

3.4.1 Adequate Data Parts

We would normally start the design with an adequate data part and add structure to it. As we add structure, some of the information in the data part is encoded in the structure and so we may simplify the data part. In this way, information is gradually moved from the data part into the structural part of the representation. When going the other way (i.e. removing the structure), information must be added to the data part so that we know what the contents of each node tells us about the state.

Since each representation is the result of applying a function (i.e. *rep*) to the state name, the contents of each node must also be a function of the state name. We shall label each node with a unique function *name* such that, in the representation of a state called x, the contents of the node labelled with a function name f, is the value denoted by $f(x)$. For example, in the representation of queues, we may label nodes as in figure 6

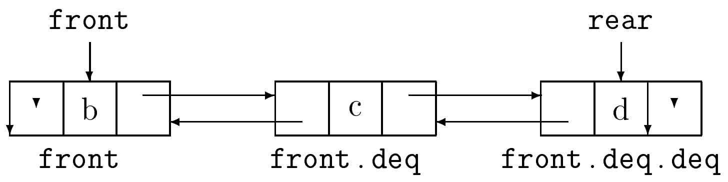


Figure 6

To put this more formally, let N be a set of node labels (i.e. function names), V the set of values that may be stored in a data cell and $F\colon N \to (X \to V)$ a naming function associating node labels with functions. We may describe the data part of the representation of a given state (wrt a given choice of F) using a function assiging values to node labels. We shall use the function $\mathit{data\text{-}rep}_F\colon X \to (N \to V)$ for this purpose and note that $\mathit{data\text{-}rep}_F\, x \subseteq \{\langle f, (F\, f)\, x\rangle \mid f \in N\}$.

In our queue example, we may name the functions using the function `deqs`:

$$\texttt{deqs} = \{\langle 0, \texttt{front}\rangle, \langle 1, \texttt{front}\,.\,\texttt{deq}\rangle, \langle 2, \texttt{front}\,.\,\texttt{deq}\,.\,\texttt{deq}\rangle, \ldots\}$$

This may be used to give the following description of the data part of the representation of queues:

$$\mathit{data\text{-}rep}_{\texttt{deqs}}\, x = \{\langle i, (\texttt{deqs}\, i)\, x\rangle \mid 0 \leq i < \texttt{length}\, x\}$$

We may readily see that this data part is adequate since, for any queue q:

$$q = [\texttt{front}\, q, \texttt{front}\,.\,\texttt{deq}\, q, \texttt{front}\,.\,\texttt{deq}\,.\,\texttt{deq}\, q, \ldots]$$

i.e. the list used to represent the state in the specification may be reconstructed from the data part of the representation.

3.4.2 Efficient Data Parts

After adequacy, our next major concern is how efficiently the data structure may be used. There are two things contributing to the efficiency (or inefficiency) of the implementation of an operation: the number of nodes accessed and the difficulty of the manipulation of the values stored in them. Often the cost of manipulating the values is insignificant compared to the cost of gathering them and so we shall emphasise the cost of accessing nodes.

From this, we derive the following definition of efficiency:

> Let A and B be implementations of a specification and let S be a set of sequences of operations (i.e. modifiers and observers). *A is more efficient than B for S* if the number of nodes accessed when executing A's implementation of S is less than the number of nodes accessed when executing B's implementation of S.

If desired, this could be generalised by assigning a weight to every sequence and comparing the weighted sums of the number of nodes accessed to execute each sequence in A and B.

In order to achieve this efficiency goal, we would expect that a good data representation would require only a small number of nodes to be accessed in order to determine the result of any observation.

In our queue example, `FRONT` need only examine one node (at most) to determine which value to return and hence is an efficient operation.

Suppose though that we add an observer to the specification which returns the length of the queue. We may use the same data part (i.e. the data part given above is still adequate) but, to determine the length of the queue, one implementation of the observer would have to count the nodes in the representation and so would be rather inefficient. However, if we added another node (which stored the length of the queue) to the representation, we could implement the `length` operation more efficiently because the desired value (i.e. the length) can be calculated by examining a small number (1 here) of nodes.

If we ignore the choice of imperative implementation and do not consider implementations which *modify* the representation of the state, implementation of the modifiers is rather similar to implementation of the observers — requiring only the calculation of the value of each node in the new representation from those in the old representation. Since all new nodes and almost all old nodes have to be accessed by implementations of the modifiers, an efficient representation would usually have a small number of nodes. (We are also less justified in ignoring the cost of calculating each value.)

However, allowing the reuse of parts of the representation of a state in the representation of its successor states makes it possible to significantly increase the efficiency of implementations, if we can avoid accessing most of the nodes in the representation being modified. Since a node may only be modified if it is accessed, this means that in an efficient representation, most of the nodes in a state x must have the same contents (and links) as nodes in the representation of its successors σx (for $\sigma \in M$). For example, figure 7 shows the effect of the modifier `DEQ` on the data part of the representation in figure 6 and the way in which the nodes in the original representation are reused for the representation of the state after applying `DEQ`.

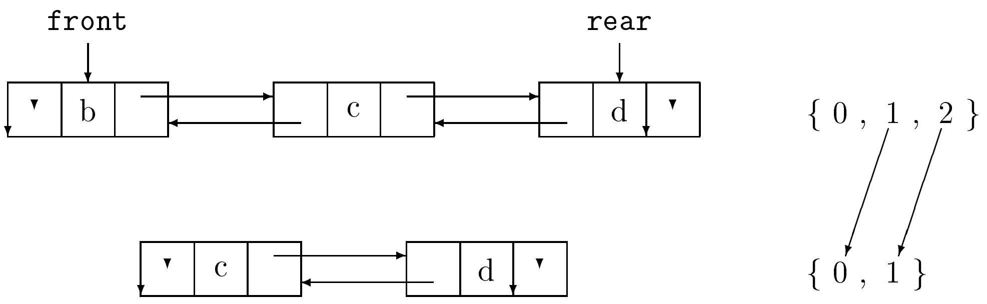


Figure 7

In this example, the choice of how to reuse nodes is obvious; in more complex examples, it is less so — often the data part must be redesigned to ensure that features of the state which change independently of each other are stored separately.

The above discussion deals with what appear to be the major issues in the design of the data part of an efficient representation:

- the data part should be adequate;
- it should be possible to calculate the result of an observation by examining a few nodes and so the representation should store features of the state closely related to what can be observed;
- it should be possible to modify a representation by accessing only a few nodes, and so the data parts of adjacent states (x and σx) should be very similar (i.e. there should be a one-to-one function $c_{x,\sigma}\colon \mathit{dom}(\mathit{data\text{-}rep}\, x) \rightarrow \mathit{dom}(\mathit{data\text{-}rep}\, \sigma x)$ which is defined for most of its domain and for which corresponding nodes have the same contents.
- there should be a description of how to calculate the successor of each representation which respects the correspondence between node labels (i.e. if $n \in \mathit{dom}(c_{x,\sigma})$, the node labelled n in the representation of x is reused as the node labelled $c_{x,\sigma}\, n$ in the representation of $\sigma\, x$.)

We shall now consider the entire design process.

3.5 Summary

To summarise our analysis, when designing a data structure one should first choose the data part and then design a structure to manipulate and access it.

The data part of the representation should initially be adequate; it should store the information required by the implementation of the observers in a small number of nodes; and it should have similar representations for adjacent states.

Starting with an adequate data part, a description of how to calculate the result of applying an observation and of how to calculate the data part of a state from its predecessor, the structural part is gradually added. As structure is designed and added, the data part may be simplified by removing nodes or labels which are no longer required for adequacy.

Then, observation points are added to the representation of each state so that they provide access to the cells from which the result of observing that state is calculated. Similarily, update points are added, thus allowing the values of new cells to be calculated and various changes to be made. After fusing entry points and eliminating those entry points which we wish to approximate, links are added in the representation of each state along the routes of the entry points. Links are fused if possible and desirable and, if necessary, the cycle repeats from the introduction of the entry points.

4 Conclusions

We have presented our ideas on how data structures can be designed. At present we recognise the following limitations in our view of the design process:

1. Although we have an abstract machine (not reported here) which may be used to describe an implementation, further work is required to determine under which circumstances the "optimisations" described here actually do result in a more efficient implementation.

2. We have no clear idea of how a good data part may be designed. This is one of our major interests at the present moment.

3. Throughout our discussion, we have considered how to implement an observation of a particular state, how to change the representation of a particular state into one of its successors, etc. Generating an implementation, requires that we generalise the implementations of each observer and of each modifier into implementations which work for any state. Our approach currently does not consider such generalisation.

4. As yet we are unable to cope adequately with problems requiring some form of search/lookup for their solution. However, since our problem seems only to be the large (usually infinite) number of entry points introduced, it seems plausible that a few techniques for reducing the number of entry points (such as use of hash tables, search trees, etc.) would largely overcome this limitation.

5. We cannot handle specifications permitting a range of possible values for a given observation. (For example, a common observer for sets is a choice operation which returns one of the elements of the set it is applied to.) This

problem can be avoided by strengthening the specification until there is only a single value for every observation. However, this solution is not entirely satisfactory since the information required to make a good (i.e. efficiency inducing) choice of how to strengthen the implementation is not available until implementation has begun.

5 Acknowledgements

This work was strongly motivated by and based on work reported in [2] and [3]. It was financially supported by an SERC Research Studentship.

References

[1] Mary E. d'Imperio, Data Structures and their Representation in Storage, in Halpern, Shaw (editors), Annual Review in Automatic Programming, Volume 5, pp. 1–76, International Tracts in Computer Science and Technology and Their Applications, Pergamon Press Ltd., 1969.

[2] Muffy H. Thomas, Implementing Algebraically Specified Abstract Data Types in an Imperative Programming Language, in TAPSOFT '87, Pisa, Italy, Lecture Notes in Computer Science, Volume 250, Springer Verlag, 1987.

[3] Muffy H. Thomas, The Imperative Implementation of Algebraic Data Types, Research Report CSC/88/R4, Computing Science Department, University of Glasgow, 1987 (also, Ph.D. thesis, University of St. Andrews, 1987.)

[4] Richard J. Bird and Philip Wadler, An introduction to Functional Programming, Prentice-Hall, 1988.

[5] Aho, Hopcroft, Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974.

[6] Donald Erwin Knuth, The Art Of Computer Programming, Volume 3, Sorting and Searching, Addison Wesley, 1973.