Implementing Algebraically Specified Abstract Data Types in an Imperative Programming Language

Muffy Thomas,
Dept. of Computing Science,
University of Stirling,
Stirling, Scotland.

Abstract

We consider one aspect of the implementation of algebraically specified ADTs: choosing data structures for an efficient implementation. The class of hierarchical ADTs which insert and access data without key is considered. The *storage relations* and *storage graphs* (relations with additional efficiency information) of an ADT are defined and we discuss how implementation decisions can be made according to their properties.

1. Introduction

The algebraic specification of abstract data types, *ADTs*, [ADJ] [EhM], encourages the construction of correct and efficient programs by separating the two concerns of *specification* and *implementation*. The specifier concentrates on problem solving and capturing the intended behaviour of the data objects. When the specifier is satisfied that the specification is in some sense good (perhaps consistent, complete and satisfies some required properties), then the specification may be implemented.

The implementer concentrates on the problems of efficient representation and storage management in the implementing language whilst ensuring correctness. The degree of difficulty of implementation is inversely related to the similarity between the specification language and the implemententation language.

There has been considerable research into methodologies and techniques for algebraic specification [Geh] [PeV], and into the implementation of ADTs by functional programs, [Moi] [Pro] [Sub]. Although students and programmers have been implementing ADTS in imperative languages for some time according to intuition and informal rules, there is little methodology and there are few software tools available to aid the implementation of ADTs correctly and efficiently.

Here we consider some of the problems of implementing ADTs in a language such as Pascal.The aim of the paper is twofold. The first is to formalise some aspects of the problems of choosing the implementing data structures; we consider choosing linked data structures for the class of hierarchical ADTs which do not insert or access data by key. The second is to discuss how an efficient implementation can be constructed using the chosen data structures .

1.1 Outline of Paper

First, we shall formalise, for a class of ADTs, the notion of *storage graphs*. A storage graph is a graph representation of an element of an ADT. Each storage graph is a directed graph with some nodes designated as *access nodes*.The edges are labelled by a *storage relation* and the nodes are labelled by a *contents set*. Given an ADT, a class of storage relations is defined. They are derived from the specification and describe the order in which the data items held in an element of an ADT may be traversed, or reached. The class of contents sets is derived from the class of storage relations. The access nodes are deduced from the implementation of the operations of the ADT as operations on storage relations.

Second, we discuss how to choose the implementing data structures according to the properties of

the storage graphs of an ADT. As an example we show how to choose the data structures for a linked implementation of a Queue specification. Finally, we briefly discuss how to construct an implementation of the operators of an ADT using the chosen data structures.

2. Related Work

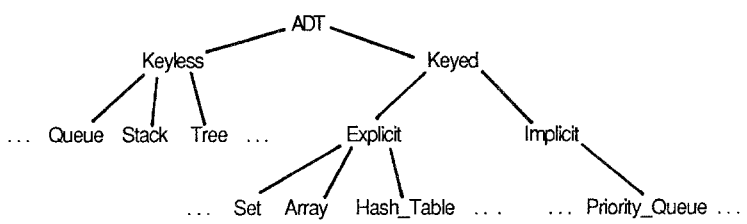
Most related work is concerned with either imperative implementations for model-based specifications [Kan] [Low] [Row] [Set], or functional implementations for equational specifications [Ape] [Moi] [Sub]. There is little related work on the problem of imperative implementations for equational specifications. The [Ape] project analyses equational specifications for a singly-linked list and doubly-linked list implementation; but because the system is knowledge-based and without a formal methodology, there are problems with the integrity of the system. The idea of representing data structures by directed graphs was first suggested in the early seventies by both [Ear] and [Ros] (the latter with a restriction to connected digraphs). One aim of the present paper is to reconsider these digraph approaches within the framework of algebraic specification and high level imperative programming languages with user defined data types.

3. Keyless Abstract Data Types

In this section we define some notation for the class of hierarchical ADTs without key and we discuss the reasons for restricting attention to this class.

Some familiarity with equational algebraic specification is assumed; for example, see [EhM] and [ADJ]. A specification consists of a signature Σ and a set E of equations. Specifications are hierarchical [Bro] and include two designated sorts, the *derived sort* and the *primitive sort*.

The figure below displays a taxonomy of ADTs: we shall consider the class of ADTs which we call *keyless ADTs*.



A keyless ADT imposes a structure on the elements of the primitive sort which is *independent* of any existing relationship between the elements. Storage and retrieval are specified by reference to some chronological ordering.

A keyed ADT imposes a structure on the elements of the primitive sort which is *dependent* on some property of the elements. If the key is explicit, the storage and retrieval are specified by reference to some relation between indices and primitive elements. If the key is implicit, then storage and retrieval is specified by reference to some ordering relation between primitive elements.

Clearly, the way in which an ADT is implemented depends on whether and how the ADT uses keys. Moreover, whereas the implementation of keyless ADTs may, in general, be constructed without reference to an algorithm or program using the ADT, the implementation of keyed ADTs is much more

dependent upon the dynamic use of the ADT. In the following, we shall consider only keyless ADTs.

Definition: A **hierarchical ADT** is a specification (Σ,E) containing at least two distinguished sorts, a **derived** sort τ and a **primitive** sort δ. The **primitive specification** [Bro] is denoted (Σ_p,E_p).

Definition: Let (Σ,E) be a hierarchical specification. (Σ,E) is **keyless** iff

i) $\forall\ \Sigma_{w,\tau}$: if $\Sigma_{w,\tau}$ is inhabited then $w \in \{\delta,\tau\}^*$,

ii) $\forall\ \Sigma_{w,\delta}$: if $\Sigma_{w,\delta}$ is inhabited then $w = \tau$,

iii) the equations in $E \backslash E_p$, where $\backslash$ is set difference, do not contain operators from Σ_p.

3.1 Partitioned Specifications

In order to discuss arbitrary specifications, an operator classification is necessary. Our classification extends the [KaS] classification of generators and defined functions.

The set of generators of a signature Σ is denoted Σ_g. We make a further subdivision of the defined operators. For example, some operators, such as **pop** in the usual **Stack** specification can be defined as "eliminators", and some, such as **top**, as "selectors". We formalise this distinction by designating disjoint sets of operators in a hierarchical specification.

Definition: A **partitioned** specification (Σ,E) is a hierarchical, keyless specification in which the operators of Σ are partitioned into 6 classes: Σ_{gd}, Σ_{gp}, Σ_e, Σ_r, Σ_s, and Σ_o. The arities of the operators in each class are constrained as follows:

$\Sigma_{gd} \subseteq \Sigma_{w,\tau}$ where $w \in \{\delta,\tau\}$ $\qquad$ Σ_{gp} = generators of Σ_p

$\Sigma_s = \Sigma_{\tau,\delta}$ $\qquad$ $\Sigma_e \subseteq \Sigma_{\tau,\tau}$

$\Sigma_r \subseteq \Sigma_{\tau,\tau}$ $\qquad$ $\Sigma_g = \Sigma_{gd} \cup \Sigma_{gp}$.

The partition classes denote the **generator** (derived), **generator** (primitive), **eliminator**, **rearranging**, **selector** and any **other** operators resp. The partition is given by the user; we now explain the ideas behind the definition. Generators define all the values of the specification, terms containing only generators and variables are called **generator terms.** A **selector** is an operator which returns elements of the primitive type. An **eliminator** is an operator which eliminates, or removes, elements of the primitive type contained in a generator term of the derived type. For example, the operator **tail** is an eliminator in the usual specification of lists; the result of an application of **tail** to a list generator term is always another list which contains fewer terms of the primitive sort. A **rearranging operator** is an operator which preserves the primitive type elements contained in a generator term of the derived type. For example, the operator **reverse** is a rearranging operator in the usual specification of lists; the result of an application of **reverse** to a list generator term is always another list containing exactly the same primitive terms.The remaining operators are in the "others" partition. This may include predicates and any operators which add primitive type elements to derived type elements but which are not designated as generators.

3.2 Semantic Requirements

In order to prove the classifications given in later sections, it will be necessary to ensure that there is at least one ground generator term in each equivalence class induced by the equations. Therefore specifications should be consistent and complete [KaS]. Completeness means that every term that is the application of a defined function to a ground generator term can be shown to be equivalent to a ground generator term. Errors are allowed, but do not propogate. The constant *err* is included in all primitive signatures, it is an *improper* element; all other elements are *proper*. In the following, specifications are partitioned, keyless, consistent and complete.

The following specification of **Queue** will be used to illustrate the subsequent definitions. We omit mention of the primitive operations on **Nat**.

spec	**Queue**		
basedon	Nat		
sorts	queue		
opns		***eqns*** ∀ q:queue, d,d':nat.	
eq:	queue	front(eq)	= *err*
add:	queue,nat-> queue	dequeue(eq)	= eq
front:	queue -> nat	dequeue(add(eq,d))	= eq
dequeue:	queue -> queue	dequeue(add(add(q,d),d'))	= add(dequeue(add(q,d)),d')
isempty:	queue -> bool	front(add(eq,d))	= d
		front(add(add(q,d),d'))	= front(add(q,d))
		isempty(eq)	= true
		isempty(add(q,d))	= false

partition Σ_{gd} = {eq,add} Σ_e = {dequeue} Σ_s = {front} Σ_r = {} Σ_o = {isempty}

4. Linked Data Structures

Data structures in languages such as Pascal are classified by storage allocation mechanism, namely static or dynamic. Implementation methods are classified accordingly. Array based implementation , or *sequential* allocation, exploits the fact that the index type is ordered. If arithmetic operations are also available in the index type, then related items may be stored at positions whose difference is defined by an arithmetic expression. In contrast, direct implementation, or *linked* allocation, links cells together explicitly. Because the positions of free cells are not related, (they are removed one at a time from the heap at runtime), cells containing related items must be explicitly linked together. Linked implementations use only as much space as is needed (apart from the overhead of links) whereas sequential implementations may waste space. However, sequential allocation may be preferable when specifications are bounded, or when keyed (random) access to stored items is required. Because our specifications are not bounded and specify keyless ADTs, only linked implementations are considered.

A linked implementation requires the definition of a **data cell** data structure and a **head cell** data structure.

Data cells contain the "data", ie. elements of the primitive sort; together they form an *implementation structure* (for example, a singly-linked list). The implementer must be able to deduce from the axioms which data cells should be linked together; namely, which primitive type elements should be related. For example, we must decide, for a binary tree, whether we should be able to retrieve the child of a node, or the parent of a node, or both, efficiently. A methodology for deriving implementation structures is

required.

Data cells are only accessed through a head cell. The head cell represents the element of the derived type by holding the address, or index, of one or more positions in the implementation structure. The nature and number of positions held in a head cell can affect program efficiency; an additional location in the head cell can reduce the time complexity of several procedures. There are no fixed rules for determining which positions the head cell must refer to and we just rely on experience and reasoning. [Mar] calls such positions "naturally designated positions". For example, we would include the top position in a stack head cell, or the root, and possibly the leaves, in a tree head cell. Clearly the choice of designated, or *access,* positions depends on both the underlying implementation structure and the operations of the ADT; a methodology for deriving access positions from implementation structures and operations is required.

5. Storage Relations

In this section we define the storage relations of an ADT. A storage relation describes the way in which primitive sorted terms, the "data", are removed and selected from a term of the derived sort, the "data structure". The relation incorporates some implementation decisions because, in general, there are several ways of deriving a particular term of the primitive sort from a term of the derived sort. We define some properties of storage relations which are useful from an implementation point of view and show how an ADT can be classified according to its storage relations.

Given a term t, of the derived sort, a particular term d, of the primitive sort, may be retrieved by applying various permutations of rearrangers, eliminators, and a selector. Rearrangers and eliminators may be arbitrarily interleaved; the application of the selector must of course be last. In general, d may be described (if possible) by a term of the form: $\sigma_s(\sigma_n(...(\sigma_1(t)...))$ where $\forall i: 0 \le i \le n: \sigma_i \in (\Sigma_r \cup \Sigma_e)$, $\sigma_s \in \Sigma_s$. In many specifications there will be several possible choices for $\sigma_1,...,\sigma_n$. We will restrict our attention to the following possibilities: $\sigma_1 \in (\Sigma_r \cup \Sigma_e)$ and $\forall i: 1<i\le n: \sigma_i \in \Sigma_e$.

The motivation for this restriction is as follows. The storage relations reflect a view of how data is stored and retrieved in an element of the ADT; we look for the simplest structure which allows efficient retrieval of the stored data. We therefore consider, for every t of the derived sort, how the data it contains may be retrieved (by elimination and selection), and how, after the rearrangement of t, the data is retrieved (by elimination and selection). For each term of the primitive sort there may be one, many, or no terms describing its retrieval from a term of the derived sort. The storage relation defines the order in which primitive terms are retrieved given these restrictions. The relation ensures that the efficiency of selection, (repeated) elimination and rearrangement are taken into account; the efficiency of repeated rearrangement is not ensured, if it is intended then the user should define a new operator.

5.1 Terms with Variables

We are not concerned with the *values* of the terms of the primitive sort as such, but with their *positions* in ground generator terms of the derived sort. We define a new signature for each specification; the signature contains δ-sorted variables in place of the primitive signature.

Definition: Let (Σ,E) be a specification with primitive signature Σ_p. Let X be an infinite set of δ-sorted variables distinct from those occurring in E. Let Σ^* denote $(\Sigma \setminus \Sigma_p \cup X \cup err)$; the elements of X are now considered as constants of sort δ. The partition of Σ is given by taking $(\Sigma^*)_g$ to be $(\Sigma_{gp} \cup X \cup err)$; the other partition classes are as before.

The equations in E may be regarded as Σ^*-equations, since no operators from Σ_p occur in E [section 3]. We may therefore consider the quotient term algebra $T(\Sigma^*)/\cong_E$ as a (Σ,E)-algebra.

Because we want the δ-sorted constants to denote positions in τ-sorted ground generator terms, we will consider only those congruence classes which contain ground generator terms with at most one occurrence of each δ-sorted constant.

Definition: Let (Σ,E) be a specification.

$T^*(\Sigma,E) =_{def} \{ C \in T(\Sigma^*)/\cong_E \mid$ if $t \in C$ and $t \in T((\Sigma^*)_g)$ then t contains at most one occurrence of each δ-sorted constant$\}$

Some examples will illustrate this definition. If (Σ,E) is the **Queue** specification and **X={x1,x2,...}**, then$T^*(\Sigma,E)$ contains classes such as: **[add(eq,x1)]** and **[dequeue(add(add(eq,x2),x2))]**. It does not contain the class **[add(add(eq,x1),x1)]**. If (Σ,E) is the sequence specification **Seq** given in [Bro], then $T^*(\Sigma,E)$ contains classes such as:

[conc(m(x1),m(x2))], **[conc(m(x3),m(x4))]**, and **[conc(conc(m(x1),m(x2)),conc(m(x3),m(x4)))]**.

It does not contain the class **[conc(conc(m(x1),m(x2)),conc(m(x1),m(x2)))]**; the function defined by **conc** is partially defined on $T^*(\Sigma,E)$.

5.2 Relations

Given a term t, we define the *storage relation* at t; we begin by defining $\downarrow t$, a subset of $T^*(\Sigma,E)_\tau$.

Definition: Let (Σ,E) be a specification and let $t \in T(\Sigma^*)_\tau$.

$\downarrow t =_{def} \{ C \in T^*(\Sigma,E)_\tau \mid \exists n \geq 0 : \exists \sigma_1,...,\sigma_n \in \Sigma_e : \exists t' \in T(\Sigma^*)_\tau :$
$((\,[t] = [t']\,) \vee (\,\exists \sigma_r \in \Sigma_r : [t'] = [\sigma_r(t)]\,)) \wedge (\,\sigma_n(...\sigma_1(t')...) \in C\,)\}.$

In the Queue example, given t = **[add(add(eq,x1),x2)]**, $\downarrow t$ = { **[add(add(eq,x1),x2)]**, **[add(eq,x2)]**,**[eq]** } ; $\downarrow t$ consists of the classes containing the subqueues of t.

Definition: Let (Σ,E) be a specification , and let $t \in T(\Sigma^*)_\tau$. The **elimination** relation --> on $\downarrow t$ is defined by

$C \dashrightarrow C' =_{def} \exists \sigma \in \Sigma_e : \sigma(C) = C'$, for $C,C' \in \downarrow t$.

It is important to note that --> does not denote the (syntactic) sub-term relation, although it may

coincide with it in some specifications such as **Stack**. In the **Queue** example,

[add(add(eq,x1),x2)] --> **[add(eq,x2)]**

because **dequeue ([(add(add(eq,x1),x2))]) = [add(eq,x2)].**

We now use the selectors and --> to construct a family of relations on $T^*(\Sigma,E)_\delta$.

Definition: Let (Σ,E) be a specification, and let $t \in T^*(\Sigma)_\tau$. The **storage relation** at t, $\Rightarrow_t$, is the following binary relation:

$D \Rightarrow_t D' =_{def} \exists \sigma \in \Sigma_s$: $\exists C,C' \in \Downarrow t$:

$((C \text{-->} C') \wedge (\sigma(C) = D) \wedge (\sigma(C') = D'))$ for $D, D' \in T^*(\Sigma,E)_\delta$.

The interpretation of $\Rightarrow_t$ depends on the specification. For example, in the usual **Stack** specification $\Rightarrow_t$ denotes "after"; $x \Rightarrow_t y$ means that x was put on the stack after y and is therefore more accessible. In the **Queue** specification, $\Rightarrow_t$ denotes the converse, ie. "before"; $x \Rightarrow_t y$ means that x was put on the queue before y and is therefore more accessible. In the **Queue** example, given $t =$ **add(add(eq,x1),x2)**, there is only one proper pair in $\Rightarrow_t$, namely **[x1]** $\Rightarrow_t$ **[x2]**.

We will restrict the domain of $\Rightarrow_t$ to the proper "contents" of [t]. Recall the notation $f^{->}(S)$ for the image of S under f.

Definition: Let (Σ,E) be a specification and $t \in T(\Sigma^*)_\tau$. The **contents set** of t, $\Downarrow t$, is defined as follows:

$\Downarrow t =_{def} \cup \{ \sigma^{->}(\downarrow t) \mid \sigma \in \Sigma_s \wedge \forall c \in C.\ c \text{ is proper}\}$.

As an example, consider the specification of **Stack** with **pop2**, an operator which removes two items at a time. Given $t =$ **push(push(push(push(create,x1),x2),x3),x4)**, then $\Downarrow t = \{$ **[x4], [x2]** $\}$.

5.3 Properties of Relations

We will, in the following sections, make implementation decisions based on properties of the structures defined by storage relations on contents sets. The implementer must use his or her imagination to decide which properties might be useful for the implementation; we define some such properties below.

Various conditions may be imposed on a relation on a set and its elements; the following conditions from [End] are standard: **reflexive**, **transitive**, **symmetric**, **antisymmetric**, **comparable**, **minimal**, and **maximal** We define some further conditions.

Definition:

1. R^* is the reflexive, transitive closure of R.
2. A relation R on set S is **down-directed** iff $\forall x,y: \exists w$: $(x\ R^*\ w \wedge y\ R^*\ w)$.
3. A relation R on set S is **upwards-directed** iff $\forall x,y$: $\exists w$: $(w\ R^*\ x \wedge w\ R^*\ y)$.
4. A relation R on set S is **n-regular** iff every element is related to no elements, or to exactly n distinct elements.

5. A relation R on set S is **(n:m)-regular** iff it is not p-regular, for some p, $n \leq p \leq m$, and every element is related to no elements, or to no more than m elements and no less than n elements.
6. A relation R on set S is **singly-linked linear** iff R^* on S is antisymmetric, all pairs in R^* are comparable and when S is non-empty, minimal and maximal elements exist.
7. A relation R on set S is **singly-linked circular** iff R on S is antisymmetric and 1-regular, and R^* on S is symmetric.
8. A relation R on set S is **doubly-linked linear** iff R on S is symmetric, R^* on S is symmetric, all pairs in R^* are comparable, and R on S i s (1:2)- regular.
9. A relation R on set S is **singly-linked down-directed** iff R^* on S is antisymmetric and R on S is down-directed.
10. A relation R on set S is **doubly-linked circular** iff R on S is symmetric, all pairs in R^* are comparable, and R on S is 2-regular.

We use the properties of storage relations to classify ADTs:

Definition: Let (Σ,E) be a specification. When for every $t \varepsilon T(\Sigma^*)_\tau$, $(\Downarrow t, \Rightarrow_t)$ has the property X (of being singly-linked linear etc.), we say that (Σ,E) has **storage type** X.

For example, we can show that **Queue** has a singly-linked linear storage type; the usual specification of **Stack** also has this storage type. As further examples, consider the sequence specification **Seq**, the usual specification for **Binary_Tree** and the usual specification for **List**; these types have doubly-linked linear, 2-regular singly-linked down-directed and singly-linked linear storage types resp. If we add a circular **shift** operator to **List** ,the storage type becomes singly-linked circular ; if we add a **reverse** operator (either using an append operator or using an auxiliary binary operator) then we have a doubly-linked linear storage type. These results are not suprising; we would expect to implement stacks and ordinary lists by similar data structures but we would not expect to implement reversible lists efficiently with the same data structure. The proofs of these classifications have been done manually; mechanisation of these proofs is planned [Sti].

6. Storage Graphs

In this section the *storage graphs* of an ADT are defined. A storage graph is a *representation* of an element of an ADT; it is a directed graph with some additional information about which nodes should be accessed efficiently at any time.

Definition: A **storage graph** is a triple (N,E,A) where (N,E) is a directed graph with **nodes** N and **edges** E , and A is a non-empty subset of N whose elements are calles **access nodes** .

Storage graphs will be defined for all classes $[t] \varepsilon T^*(\Sigma,E)_\tau$. The nodes and edges are given by the storage relations and contents sets; it remains for us to define the access nodes.

6.1 Access Nodes

The function of the set of access nodes is twofold. FIrst, it defines the access to nodes in the digraph by indicating which nodes are immediately accessible at all times. Clearly all nodes should be reachable; namely, for each node in a storage graph, there should be an access node such that there is a path from the access node to that node. (This is similar to the notion of a root ,or countable basis [Har] except that the set is not required to be minimal.) Second, the set defines the space-time trade-off; namely, membership of this set may be allocated to a position which is not necessary to ensure reachability but would enable (time) efficient implementations of certain operations. We proceed to define the access nodes according to these two principles.

6.1.1 Accessibility

The most "natural" access to the nodes of the digraph is that which is defined by the selectors. Clearly designating selected nodes as access nodes ensures that selectors can be implemented in constant time. However, these nodes alone do not ensure that all nodes are reachable, the nodes selected after one application of rearrangement must be included. Together, these nodes are referred to as the *selected positions*, or SP(t) given $t \varepsilon T(\Sigma^*)_\tau$.

> **Definition:** Let (Σ,E) be a specification with selectors $s_1, \ldots, s_n$ and rearrangers $r_1, \ldots, r_m$. Let $t \varepsilon T(\Sigma^*)_\tau$, the **selected positions** of t, **SP(t)**, are defined by
> $SP(t) =_{def} \{ s_1(t), \ldots, s_n(t), s_1(r_1(t)), \ldots, s_n(r_1(t)), \ldots s_1(r_m(t)), \ldots, s_n(r_m(t)) \}$.

The selected positions ensure that all nodes are reachable.

Lemma: Let (Σ,E) be a specification . $\forall\ t \varepsilon T(\Sigma^*)_\tau$: $\forall x\ \varepsilon \Downarrow t : \exists p \varepsilon SP(t)$: $(p \overset{*}{\Rightarrow}_t x)$.
Proof: By defn. of $\Downarrow t$, there is a σ_s in Σ_s and C in $\downarrow t$ st. $\sigma_s(C)=x$. By defn. of $\downarrow t$, there is an $n \geq 0$, $\sigma_1,\ldots,\sigma_n$ in Σ_e, and t' in $\downarrow t$ st. $\sigma_n(\ldots\sigma_1(t')\ldots)$ is in C . Either $[t] = [t']$, or there is a σ_r in Σ_r st. $[t']=[\sigma_r(t)]$. By defn. of --> on $\downarrow t$, $[\sigma_{n-1}(\ldots\sigma_1(t')\ldots)]$ --> C, $[\sigma_{n-2}(\ldots\sigma_1(t')\ldots)]$ --> $[\sigma_{n-1}(\ldots\sigma_1(t')\ldots)]$, ... , $[t']$ --> $[\sigma_1(t')]$; by transitivity, $[t'] \overset{*}{\rightarrow}$ C. By the (transitive) defn. of $\overset{*}{\Rightarrow}_t$, $[\sigma_s(t')] \overset{*}{\Rightarrow}_t \sigma_s(C)$. $\sigma_s(C)=x$, and so $[\sigma_s(t')] \overset{*}{\Rightarrow}_t x$.
$SP(t)=\{ [\sigma_s(t)], [\sigma_s(\sigma_r(t))] \}$; if $[t']=[t]$ then $p= [\sigma_s(t)]$ otherwise $p= [\sigma_s(\sigma_r(t'))]$.

6.1.2 Space-Time Tradeoff

The selected positions define the nodes which can be designated as access nodes; which other positions, in general, should be added for efficient implementations? The answer depends on the structure of the storage relation *and* the operations of the ADT. For example, both **Stack** and **Queue** have singly-linked linear storage type. The selected positions, in both cases, are the maximal positions, given by **top(t)** and **front(t)** resp., for some t. After inspecting the operations of these ADTs, we would expect, in the **Queue** example, to include the minimal position as an additional access node; we would not expect to include this position among the access nodes of **Stack**. Additional access nodes will be

defined by the (additional) selectors which must be synthesised in order to implement the specification by storage relations. The set of such selectors will called **ST(t)**, for some $t \in T(\Sigma^*)_\tau$

A hierarchical, algebraic notion of implementation is adopted [Gog] [Nou]. Implementation is essentially the process of imposing the structure of the (initial) algebra of the *implemented* ADT onto the (initial) algebra of the *implementing* ADT. The operators of the implemented ADT may be implemented by *derived* operators in the implementing ADT; techniques such as those in [KaS] allow us, under certain circumstances, to synthesise derived operators automatically.

An algebraic specification of storage relations and an abstraction mapping between the storage relations and the τ-sorted elements is required. The specification depends on the specification of sets and relations, and also on the partition of the object specification. For the purposes of this paper we do not consider how to give a parameterised specification of storage relations, nor do we give the entire specification for the **Queue** example. Instead, the relevant signatures are given; the set theoretic equations are obvious and the others can be derived from the definitions in section 5.2. Specifications are parameterised using a notation like that of **Clear** [San]; comments follow after "!".

```
meta  U = sorts elem end

meta  Spec = sorts  τ, δ end

proc  Set ( D:U ) = enrich  D + Bool by
data sorts  set
opns
∅   :     set                                  ! empty
{_}  :    elem          -> set                 ! singleton
_∪_ :     set,set       -> set                 ! union
_\_ :     set,set       -> set                 ! difference
_ε_ :     elem,set      -> bool                ! membership
_==_ :    elem,elem     -> bool                ! equality
eqns      {omitted}
end       ! Set

proc  Pair ( D:U ) = enrich  D by
data sorts  pair
opns
( _,_ ):  elem,elem     -> pair                ! mk pair
s       :  pair         -> elem                ! source element
t       :  pair         -> elem                ! target element
eqns      {omitted}
end       ! Pair

Proc  Edges ( D:U ) = enrich  derive from Set( D )   by nset is set, ∅n  is ∅, [_]i s {_}   end
                      +       derive from Set ( Pair( D ) [ elem is pair ])
                                                     by eset is set, ∪ is ∪, \\ is \, ∅e  is ∅end
                      by
opns
sce  :  eset,elem -> eset          ! only pairs with elem as source
tgt  :  eset,elem -> eset          ! only pairs with elem as target
maps:   eset      -> nset          ! map s
mapt :  eset      -> nset          ! map t
eqns
all S:eset, x:pair, y:elem.    sce(S ∪ {x}, y)   = sce(S,y) ∪ {x}     if (s(x) == y)
all S:eset, x:pair, y:elem.    sce(S ∪ {x}, y)   = sce(S,y)           if (¬(s(x) == y))
all S:eset, y:elem.            sce( ∅,y)         = ∅
        {rest omitted}
end     ! Edges
```

proc Graph (S:Spec) = ***enrich*** Edges (S [elem ***is*** δ])
+ ***derive from*** Set (S [elem is τ]) ***by*** tset is set, ... ***end***
by

data sorts graph
opns

⇓	: τ	–>	nset
⇒	: τ	–>	eset
(_,_)	: nset,eset	->	graph
↓	: τ	–>	tset
_==_	: τ,τ	->	bool
_==_	: δ,δ	->	bool

{rest of signature omitted}
eqns {omitted, equations depend on partition of S}
end ! Graph

Note that storage relations are now defined on all elements of $T(\Sigma,E)/\cong_E$. Because the Graph specification contains the object specification, the generator terms of S are subterms of the generator terms of Graph(S), and the abstraction mapping comes for "free":

abs: Graph(s) -> S **abs** (⇓t ,⇒t) $=_{def}$ t.

A small constraint is imposed on the form of the specification of the derived operators (the implementations of the operators of the object specification). In every equation of the form:

$$F(\Downarrow t, \Rightarrow_t) = \text{r.h.s.}$$

r.h.s. may not contain an occurrence of a subterm of t. The motivation for this restriction is that ultimately, τ-sorted terms will be implemented by variables of type head cell (in a given environment and store); therefore definitions should not depend of the syntactic form of termsof the given ADT.

Consider, as an example, the implementation of **Queue** by **Graph(Queue).** The abstraction mapping **abs** is given by the two equations:

abs (⇓eq ,⇒eq) = eq, and **abs**(⇒add(q,d),⇓add(q,d)) = add(q,d).

The operators of **Queue** are implemented by the following derived operators (in bold upper case) in **Graph(Queue).**

∀τ:queue,D:nat.

EQ	=	(∅, ∅)	
ADD ((⇓t ,⇒t), D)	=	(⇓t ∪ { D }, ⇒t)	if (isempty(t))
ADD ((⇓t ,⇒t) ,D)	=	(⇓t ∪ { D }, ⇒t ∪ { (last(t), D) })	if ¬(isempty(t))
DEQUEUE (⇓t ,⇒t)	=	(⇓t \ { front(t) }, ⇒t \\ sce(⇒t, front(t)))	
FRONT (⇓t ,⇒t)	=	front(t)	
ISEMPTY (⇓t ,⇒t)	=	isempty	

where last is a derived operator in **Queue** with equations last(eq) = *err* and last (add(q,d)) = d .

In this example, SP(t) = { front(t) } , and ST(t) = { last(t) }.

Definition: Let (Σ,E) be a specification and let $t \in T(\Sigma^*)_\tau$. The **storage graph** at t is defined by the tuple: (⇓t , $\Rightarrow_t$, A(t)), where A(t) = SP(t) ∪ ST(t).

7. Implementing Abstract Data Types

In this section we discuss how the storage type and storage graphs of an ADT can determine the choice of implementing data structures . We briefly discuss how implementations are constructed .

7.1 Choosing Data Structures

In the absence of further information concerning the dynamic use of the ADT, the motivation for the choice of (linked) data structures is that each τ-sorted element is represented by a storage graph . The nodes of the graph are represented by data cells, the edges by pointers between data cells, and the access nodes by a head cell.

The *data type* of head cells is chosen according to the storage graphs of the ADT. The data type is a product, or Pascal ***record*** of pointers of type data cell. The number of pointer fields is defined according to the cardinality of the access node sets in the storage graphs. Namely, for each term t, there will be a one-one correspondence between the elements of A(t) and the fields of the head cell. (Note that in Pascal a record type containing exactly one field whose type is ^T, where T is some data type, is equivalent to the type ^T.)

The *data type* of data cells is chosen according to the storage type of the ADT. The data type is a **record** consisting of one field whose type is the (representation) of the primitive sort, and when the storage type is m- or (n:m)-regular, it contains m pointers of type data cell. (If the storage type is not regular then another record type is necessary in order to link together data cell pointers. We do not pursue this as we believe that ADTs with non-regular storage types cannot be specified as keyless ADTs.)

Consider the data structures for **Stack** and **Queue**. Both specifications have the same storage type, singly-linked linear; therefore for both the data cell is

data_cell = **record** contents: integer; next: ^data_cell; **end**

In **Stack**, the cardinality of the access node sets is at most 1 and in **Queue** it is at most 2.The head cell for **Stack** is ^data_cell and for **Queue** it is **record** first:^data_cell; last:^data_cell; **end.**

Consider the **Seq** example; it has doubly-linked linear storage type and the cardinality of the access nodes sets is at most 2, (there are no synthesised selectors in the Graph implementation and for all t , ST(t) = { first(t), last(t) } . The implementing data structures are:

data _cell = **record** contents: integer; succ: ^data_cell; pred: ^data_cell; **end;**

head_cell = **record** first: ^data_cell; last: ^data_cell; **end;**

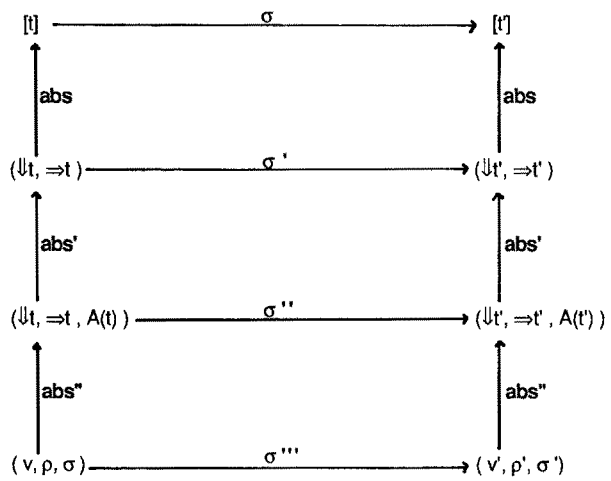
7.2 Constructing an Imperative Implementation

The details of implementation cannot be presented here and so only a brief outline follows.

It is not difficult to see that the data structures given in the previous section allow time-efficient implementations to be constructed. Of course the data structures alone do not ensure correct implementation, many properties of the storage relations, for example antisymmetry, are not ensured by the pointer types of Pascal.

The programming language, with the appropriate definitions of head_cell and data_cell, must be specified as an ADT; assume the specification is called **ProgLang**. For a given ADT **S**, the operations of **Graph(S)** must be implemented (in the usual way) by derived operators in **ProgLang**. These operators may be regarded as procedures. Elements of **ProgLang** are triples (v,ρ,σ) where v is a (head cell)

identifier, ρ is an environment, and σ a store. Implementation can be summarised by the following (commutative) diagram; assume $[t] \varepsilon T(\Sigma)/\cong_E$ and $\sigma[t] = [t']$.



In the **Queue** example, if σ is **add**, then σ' is the **ADD** given in section 6.1.2. σ", **ADD"** , is defined by

ADD"((⇓t, ⇒t, A(t)), D) = (**ADD**(⇓t, ⇒t), { front(t) } ∪ { D }) if ¬ (isempty(t))

ADD"((⇓t, ⇒t, A(t)), D) = (**ADD**(⇓t, ⇒t), { D }) if (isempty(t))

The definition of **abs"** is not as straightforward as **abs** and **abs'**. The domain of **abs"** has to be defined by giving a representation invariant [KaS] to ensure that head cells point to valid queues; the mapping itself is quite complicated because of the nature of imperative languages with pointers.

Finally, a brief note on efficiency. First, transformations in the programming language may further improve the efficiency of the implementation. For example, the head cell of a reversible list may also contain information about how the storage relation is represented by the fields of the data cells. Second, we may wish, in some circumstances, to "trade" back some space for time; namely, to make the access node set a countable basis. In the implementation, there is a time overhead associated with each access node; namely, the head cell has to be adjusted according to changes to the implementation structure. In some cases this overhead may outweigh the benefits of constant time access to some position(s); then it would be preferable to remove the position(s) from the access node set (thus making access to them linear).

8. Conclusions and Future Work

We have formalised some aspects of the efficient implementation of ADTs using imperative data structures. Our approach agrees with intuition; when it is applied to familiar examples it produces expected results. The approach is influenced by two factors: the properties which the implementer uses to classify ADTs, and the partitioning of signatures. The properties should be useful from the implementation point of view and the partitioning must be sensible.

The investigation into a methodology for imperative implementations of ADTs has only begun and much work remains to be done. Several topics require further formalisation and the application of the definitions must be considered; it is hoped that the classification of ADTs by storage type may be proved/disproved with the help of an equational reasoning laboratory such as ERIL [Dic]. Some additional properties of the eliminator and rearranging partition classes may be required in order to prove properties such as the well-foundedness of storage relations. A large library of storage types would be useful, and the approach should be extended to include other classes of ADTs.

Acknowledgements

I would like to thank Roy Dyckhoff, Alan Hamilton, Moira Norrie, Chic Rattray and Don Sannella for many discussions and helpful comments on the topic of this paper.

References

[ADJ] Goguen J.A.,Thatcher J.W.,Wagner E.G., "ADJ: An Initial Approach to the Specification, Correctness and Implementation of Abstract Data Types", *Current Trends in Programming Methodology*, Chapter 5, 1978.

[Ape] Bartels U.,Olthoff W.,Raulefs P., "APE: An Expert System for Automatic Programming from Abstract Specifications of Data Types and Algorithms." MEMO SEKI-BN-81-01, Institut fur Informatik, Universitat Kaiserslautern, 1981.

[Bro] Broy M., "Algebraic Methods for Program Construction:The Project CIP", pg. 199-222, *Nato ASI Series vol. F8,* Program Transformation and Programming Environments, Springer-Verlag 1984.

[Dic] Dick, A.J.J. , "Equational Reasoning and Rewrite Sysytems on a Lattice of Types," PhD. Thesis, Dept. of Computing, Imperial College, London,1986.

[Ear] Earley J., "Toward and Understanding of Data Structures",*C.A.C.M.* vol. 14, no.10, 1971.

[EhM] Ehrig H.,Mahr B., *Fundamentals of Algebraic Specification 1*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1985.

[End] Enderton H.B., *Elements of Set Theory*, Academic Press 1977.

[Geh] Gehani N., McGettrick A.D (Eds.), Software Specification Techniques, Addison-Wesley 1986.

[Har] Harary F., *Graph Theory* , Addison-Wesley, Reading, Mass.,1969.

[Kan] Kant E., Barstow D.R.,"The Refinement Paradigm:The Interaction of Coding and Efficiency Knowledge in Program Synthesis", *IEEE Trans. on Soft. Eng.*,vol SE-7, no.5, Sept. 1981.

[KaS] Kapur D.,Srivas M., "A Rewrite Rule Based Approach for Synthesising Abstract Data Types, *CAAP'85"*, Lecture Notes in Computer Science ,vol. 185,Springer-Verlag.

[Low] Low J.R., "Automatic Data Structure Selection:An Example and Overview" ,*CACM* ,vol. 21 no. 5, May 1978.

[Mar] Martin J., *Data Types and Data Structures*, Prentice Hall 1986.

[Moi] Moitra A., "Direct Implementation of Algebraic Specification of Abstract Data Types", *IEEE Trans. on Soft. Eng.*, vol SE-8, no. 1, Jan. 1982.

[Nou] Nourani C.F., "Abstract Implementations and Their Correctness Proofs", *J.A.C.M.*, vol. 30, no.2, 1983.

[PeV] Pequeno T.H.C., Veloso P.A.S., "Do Not Write More Axioms Than You Have To", Proceedings of Intermational Computer Symposium, Vol 1., 1978.

[Pro] Prospectra Project Summary, ESPRIT Project no. 390.

[Ros] Rosenberg, A.L., "Symmetries in data graphs", *SIAM J.Comput.* 1, 1972.

[Row] Rowe L.A., Tong F.M., "Automating the Selection of Implementation Structures", *IEEE Trans. onSoft. Eng.*, vol SE-4, no. 6, Nov 1978.

[San] Sannella D., "A Set-Theoretic Semantics for Clear", Acta Informatica, No. 21, 1984.

[Set] Freudenberger S.M.,Schwartz J.T., Sharir M., "Experience with the SETL Optimizer," *ACM ToPLaS*, vol. 5, No. 1, Jan. 1983.

[Sti] Alvey Project No. 007, Dept. of Computing Stirling,Stirling University, Stirling Scotland.

[Sub] Subrahmanyam P.A., "A Basis for a Theory of Program Synthesis", Phd. Thesis extract, USC Information Sciences Inst. and Dept of Computer Science, Univ. of Utah, 1980.