-- --

Real-time Simulation of a Set Machine on a RAM

Robert Paige [1]

Computer Science Department
New York University/ Courant Institute
New York, NY 10012
paige@cs.nyu.edu

ABSTRACT

The analysis of set-based programs is sometimes facilitated by the computational model of a set machine; i.e., a uniform cost sequential RAM augmented with an assortment of primitives on finite sets, under the assumption that associative operations, e.g., set membership, take unit time. In this paper we give broad sufficient conditions in which to simulate a set machine on a RAM (without set primitives) in real time. Two variants of a RAM are considered. One allows for pointer and cursor access. The other permits only pointer access. Our translation method introduces a new programming methodology for data structure design and provides a new framework for investigating automatic data structure selection for set-based programs.

November 10, 1994

1 Part of this work was done while the author was a summer faculty at IBM T.J. Watson Research Center. This work is also partly based on research supported by the Office of Naval Research under Contract No. N00014-87-K-0461 and by Thomson-CSF/DSE under an ESPRIT contract.

Real-time Simulation of a Set Machine on a RAM

Robert Paige [1]

Computer Science Department
New York University/ Courant Institute
New York, NY 10012
paige@cs.nyu.edu

key words: data structure selection, real-time simulation, associative access, cursor access, pointer access, based representation

1. Introduction

In spite of its central place in mathematical discourse, set theory has had a relatively minor influence on programming language design. One conspicuous exception is SETL [28], a programming language that has demonstrated the benefits of finite set theory for software design and prototyping [20], program verification [6], and top-down program (and algorithm) development methodology [7, 24, 25]. At the same time, a decade-long experience with SETL has underscored three shortcomings in its use [11]:

(a) Standard compilation and optimization techniques fare poorly in a language with finite set theory: compiler and optimizer are complex and slow.

(b) The resulting code is often more inefficient than that produced by other very high-level languages.

(c) It is hard to perform accurate complexity analysis directly on set-based code.

It would seem that these shortcomings have kept more language designers from introducing set theory primitives into new languages. One common explanation for these shortcomings is that unordered sets have no obvious *a priori* representation in sequential memory, and data structures that are content-addressable are typically more expensive to use than arrays and lists. We argue here for the contrary position, specifically:

(d) The strength of set-theoretic constructs lies precisely in the fact that they can be realized by a number of different concrete data structures.

(e) Many programs that manipulate content-addressable structures (i.e. sets and mappings over arbitrary domains) can be optimized so that no such structures are actually built, and so that associative access operations are replaced by less expensive operations such as indexings and dereferences.

In this paper we show how to overcome the shortcomings (a), (b), and (c) by simulating sets and set operations on a uniform cost sequential RAM [2, 22] in real time. That is, we define a formal set machine to be a RAM augmented with an assortment of primitive operations on finite sets. Each set machine primitive operation q is associated with a worst case asymptotic set-theoretic time and space bound $O(f_q)$. We provide a transformation T that maps a set machine program P (satisfying sufficient conditions defined precisely in section 4) into RAM code $T(P)$

1 Part of this work was done while the author was a summer faculty at IBM T.J. Watson Research Center. This work is also partly based on research supported by the Office of Naval Research under Contract No. N00014-87-K-0461 and by Thomson-CSF/DSE under an ESPRIT contract.

by translating each primitive operation q in P into a functionally equivalent sequence of RAM operations with an $O(f_q)$ worst case asymptotic RAM time and space bounds. Consequently, if P has an $O(g(n))$ time and $O(h(n))$ space asymptotic worst-case complexity on a set machine, then $T(P)$ has the same asymptotic worst-case time and space bounds on a RAM. It is important to note that our simulation depends strongly on the presentation of the input data and that the cost of preconditioning the input to facilitate RAM computation is not included in our RAM complexity[3].

Two variants of a RAM are considered. The simplest is a pointer machine [18, 32], which accesses data using pointers but lacks direct access into arrays and cannot perform address arithmetic. The other augments the pointer machine capabilities with cursor access into arrays and tuples. Our simulation conditions provide a basis for studying when associative access can be replaced by easier cursor access without penalty and when cursor access can be replaced by pointer access without any asymptotic losses in time or space.

The results reported here extend the benefits of finite set theory, previously used in program verification, to the accurate analysis of program performance. It also provides an easy top-down methodology for designing data structures. Our techniques also provide a new framework with which to extend the data structure selection algorithms proposed in [24, 27] to produce structures that can be modified to a much greater degree at runtime.

This paper is organized as follows: Section 2 examines the advantages of set-based programming in software development. Section 3 introduces the set-machine and RAM models of computation. Section 4 describes how to simulate a set-based program on a RAM in real-time.

2. The advantages of set-theoretic notations in programming

Notations involving sets and mappings are useful for a number of reasons:

(a) Sets and mappings are abstract data types with a simple and intuitive axiomatization (see, e.g., [28, 31]).

(b) More complex abstract data types, such as stacks and queues, are often used solely to implement sets efficiently [2, 8, 22, 28, 33].

(c) Non-determinism appears naturally in association with set notations. The retrieval operation $\ni S$ denotes an arbitrary choice over the elements of set S. Similarly, the iteration (**for** $x \in S$) *Block(x)*...**end for** denotes a repeated execution of *Block(x)* for each distinct value of x enumerated from S in any order.

The importance of (a), (b), and (c) in programming practice is sufficiently well-known that the shortest summary will suffice here: these advantages facilitate program correctness proofs by making programs shorter, more abstract, and free from unimportant implementation details. Furthermore, as Schwartz has argued [29], they allow the programming task to be factored into two phases: one of algorithm design, and one of data structure selection, which is best tackled after the first is completed. Thus (a), (b), and (c) facilitate the application of decision-postponement, a widely advocated principle in contemporary software engineering.

Finally, the variety of data structures that can be used to implement sets and mappings means that a single abstract algorithm serves as a specification for a number of different low-level programs, each of which may be efficient over a different range of values of its input parameters. Thus, the use of set notations leads naturally to a top-down form of transformational programming in which inefficient but perspicuous indeterminate programs are mapped into efficient but complex determinate ones. (The skeptical reader might argue that any abstract-type methodology

---

3 For example, we could assume that the input data forms a set or represents a graph as an adjacency list.

will present the same advantages; we can only remark that no other abstract structures have been proposed as language primitives that have the generality and the simple semantics that set constructs offer.)

It is worth noting that the choice among several low-level implementations of the same abstract set-theoretic algorithm may require exceedingly subtle complexity analysis. In the absence of such analysis, it is still possible to experiment with various implementations and arrive at a pragmatic choice based on program monitoring until a more refined analysis can pinpoint an optimal solution. An extreme example of such decision postponement, spanning more than a decade, is exemplified by Kildall's general schema [17] for iterative computation of monotone data flow problems:

$[x_1,...x_n] := [0, 0,... 0]$;
(**while** $\exists$ i $\in$ $\{1..n\}$ | $f_i(x_1,..,x_n) > x_i$)
    $x_i := f_i(x_1,..x_n)$;
end loop ;

where the set $\{1..n\}$ enumerates the instructions in a program, and variables x_i represent program attributes defined on each instruction. The organization of the set of instructions and the order in which the existential search should proceed is unspecified in the preceding code. A number of orderings were proposed over the years (for which see [12, 14, 16, 36]) until Tarjan [34] found an optimal one.

3. Computational Models

We investigate three sequential computational machine models distinguished mainly by the following different fundamental operations used to access data.

(1) *associative access* tests whether an arbitrary value (called a *search argument*) belongs to a set and locates this value if it does;

(2) *cursor access* locates an indexed component of an array or tuple; the index argument is called a *cursor*;

(3) *pointer access* locates data according to its memory address.

We ask when can unit-time associative access be eliminated without penalty and replaced by unit-time cursor access. We also ask when can cursor access be replaced by simpler pointer access? To answer these questions we discuss the real-time simulation of an abstract set machine, which exploits associative access as its main primitive operation, on two more realistic sequential machine models with direct access capability - (i) an abstract pointer machine in which all data is accessed using pointers and pointer arithmetic is disallowed; and (ii) an abstract uniform cost RAM that has pointer as well as direct access capability.

It is convenient to use a simple programming language to model each of these machines. Within each of these languages *elementary* datatypes are unit-space data with unit-time operations. These include boolean, integer, real, and pointer datatypes. Pointers are used for storage, retrieval, and equality only and cannot be involved in any address arithmetic. A special constant denoted by Ω represents an undefined value. Any datatype that is not elementary is called *nonelementary*. Each language has conventional **while**-loops, **for**-loops, **if-then-else**'s, assignments, and input/output statements. We assume copy value semantics for assignment of elementary datatypes. Such assignments take unit-time to perform. A program consists of a single main procedure without subprocedures.

3.1. A list-based language

We model the abstract pointer machine with the APM language. The only nonelementary datatype in APM is the list. Lists can either be one-way or two-way, have a value field or not, and have optional pointers to the first and last list cells. If C is a list cell, then *C.val* denotes the value field, *C.prev* denotes a pointer to the previous cell, and *C.next* denotes a pointer to the next cell. If L is a list, then *L.first* and *L.last* denote pointers to the first and last cells. Lists can be dynamically allocated, modified, and deallocated at run-time. Records are defined as $O(1)$ length static one-way lists. We use pointer semantics for list assignment. Hence, all assignments in APM takes $O(1)$ time. Garbage collection is automatic and contributes no more than a constant factor to the worst case asymptotic time and space.

3.2. An array-based language

Cursor access is framed within the array processing language ARAM. The ARAM language is APM augmented with arrays of one or more dimension and tuples (i.e., an ordered set t of elements ordered from the first, denoted $t(1)$, to the last, denoted $t(\#t)$) with bounded growth space (for limited runtime growth). We assume static allocation for arrays and tuples. ARAM records are defined as $O(1)$ length one-dimensional arrays.

Unit-time direct access is supported for referencing the i-th component of a tuple using integer i as a *cursor* (e.g., $t(i)$) or the i,j-th component of a two dimentional array (e.g., $t(i,j)$). Based on direct access using cursors and cursor arithmetic, tuples and arrays can be initialized in $O(1)$ time using a stack and cursor array according to exercise 2.12 found in [2]. Likewise, the data in an array or tuple can be split into roughly equal parts (which is useful for divide-and-conquer strategies) in $O(1)$ time.

ARAM is more powerful than APM, since finding the i-th element of a list, initializing all the cells of a list, or splitting a list cannot be achieved in constant time in APM. Consequently, various data structures for hashing, tries, and the $O(log \log N)$ priority queue [4, 5] of Van Emde Boas cannot be implemented with as good an asymptotic performance in APM as in ARAM.

Whenever data can be stored within an array or tuple instead of a list, there are several other practical advantages that should be mentioned. There is a constant factor improvement in space, because *next* and *prev* pointers are absent in arrays and tuples. Cursors take up less space than pointers. Also, cursors are relocatable, while pointers are not. Arrays and tuples have the disadvantage of being essentially static data structures that, when constrained by limited total storage space, require static allocation in general and in the case of tuples permit only limited runtime growth if we do not want to sacrifice performance.

3.3. A set-based machine language

The set-based language ASM is ARAM plus finite dynamic sets, maps, and tuples. A set is an unordered collection of distinct values, while a tuple is an ordered collection of values. A map is a set of ordered pairs (i.e., 2-tuples); that is, a binary relation from a domain set (i.e., the set of first components of these pairs) to a range set (i.e., the set of second components). Thus, a map can be a single-valued function or a multi-valued binary relation. Sets and maps can be dynamically allocated, deallocated, and modified at run-time.

The primitive set-theoretic operations of ASM correspond closely to finite set theory [31] and SETL [28], and we will mainly use universally accepted mathematical notations. Several exceptions include the overloaded *size* operator denoted by $\#S$, which represents the cardinality of a finite set S or the number of components in a tuple or array S. Another is arbitrary choice, denoted by $\ni S$, which has the value of an arbitrary element chosen from a set S. We also use the unconventional SETL interpretation of maps as dynamic functions or multivalued binary relations. Function retrieval is denoted by $g(x)$ while multi-valued map retrieval is denoted by $g\{x\}$.

The space complexity for ASM data is defined as follows:

$$\text{space}(x) = O(1), \text{ if } x \text{ is fixed length data, e.g., integer, real, boolean;}$$
$$O(1) + \sum_{y \in x} space(y), \text{ where } x \text{ is a set, array, or tuple;}$$
$$O(1) + \text{space}(\textbf{domain } x) + \sum_{y \in \textbf{domain } x} space(x\{y\}), \text{ where } x \text{ is a map}$$

Table 1 lists some of the primitive set-theoretic ASM operations and their worst case asymptotic time complexities on the set machine. For each operation auxiliary space is bounded by execution time.

Primitive Set and Map Operations

$Q := \{\}$:	assign empty set	$O(1)$
$Q := \Omega$:	deallocate set	$O(1)$
Q **with**:= x	set element addition	$O(1)$
Q **less**:= x	set element deletion	$O(1)$
$Q \cup := T$	disjoint union	$O(1)$
$x \in Q$	set membership test	$O(1)$
$\ni Q$	arbitrary choice	$O(1)$
$\exists\, x \in Q$	test for empty set*	$O(1)$
$g\{x\}$	$\{ y : [u,y] \in g \mid u = x \}$	$O(1)$
$g(x)$	y, if $g\{x\} = \{y\}$; Ω (undefined), otherwise	$O(1)$
$g\{x\} := \{\}$	remove x from **domain** g	$O(1)$
$g\{x\}$ **with**:= y	add pair $[x,y]$ to g	$O(1)$
$g\{x\}$ **less**:= y	delete pair $[x,y]$ from g	$O(1)$
$y \in g\{x\}$	test $[x,y] \in g$	$O(1)$
$g(x) := \Omega$	remove x from **domain** g	$O(1)$
$g(x) := z$	make $g(x) = z$	$O(1)$
(**for** $x \in Q$) Block(x) **end for**;	execute Block for each element x belonging to Q**	$O(\#Q * \text{cost}(\text{Block}(x)))$
$Q := T$	copy set T to set Q	$O(1)$

TABLE 1. ASM Primitive Operations

* As a side effect x is assigned an arbitrarily chosen element of Q or the undefined value Ω if Q is empty.

** It is assumed that the for-loop iterates through a copy of Q. This allows Q to be modified within the loop without affecting loop iteration.

4. Real-Time Simulation

In this section we consider fairly general semantic conditions under which ASM programs can be simulated in real-time with ARAM or APM programs. Before stating these simulation conditions, we need to provide several definitions and notational conventions. It is convenient to assume that nonelementary objects are *exogenous*; i.e., are accessed indirectly by pointers. For example, a variable T whose value V is nonelementary is implemented as a pointer to V. Likewise, any nonelementary element (with value V) of a nonelementary object is assumed to store a pointer to V.

A variable V is *live* at a program point p, if there exists an execution path (containing no assignments to V or, in the case of lists, containing no assignments to aliases of V either) from p

to another point where V is used. A variable V that is not live at a program point p is said to be *useless* at p. An object is *live* at a program point p if it is the value of a live variable at p, is a component of a live array or tuple, is an element of a live set, is the domain or image set of a live map, is the first cell of a live list, or is pointed to by a live pointer. Any object that is not useful at a program point p is *useless* at p and can be deallocated.

Recall that ASM uses copy-value semantics for all nonelementary objects except lists. However, to perform assignments S:=T of nonelementary objects T in unit-time, we implement such assignments according to pointer semantics; i.e., so that both S and T would store pointers to the data associated with variable T. Moreover, an element addition S *with*:= T of nonelementary T adds a pointer to T to S. During execution, whenever there is more than one pointer to a nonelementary object T, we say that T is *pinned*. In case a map T is pinned, then its domain and all of its image sets are also pinned. If T is not pinned, we say that it is *unpinned*. Observe that executing initializations $T := \{\}$, deallocations T:=Ω, element deletions S **less**:= T, and assignments T:=S reduce the number of pointers to (and might even unpin) the old value of a nonelementary object T.

4.1. Initial Semantic Conditions

In discussing a method for simulating an ASM on either kind of RAM in real-time, it is useful to state initial semantic conditions that restrict the use of set-theoretic ASM primitives. These primitives operate on three kinds of sets - a set valued variable, the domain of a map, and a *nonsimple* set (i.e., a set that is a member of another set; for example, the image set $f\{x\}$ is nonsimple, because the range of f is organized as a set of image sets). It is important to note that operations on a map f are more conveniently seen as operations on the domain and range of f.

Below we divide these ASM primitives into seven basic categories:

(1) (retrieval) Primitive ASM operations that retrieve arbitrary elements from a set are the choice operation $\ni S$ and arbitrary search (**for** $x \in S$) **block**(x) **end if**, where S could be a set, the domain of a map, or an image set in the range of a map.

(2) (initialization) The primitive $Q := \{\}$ can be used to assign the empty set to an undefined set or map Q. When Q is a map, this operation essentially initializes both its domain and range. If Q is defined before executing this primitive, the operation is called *reinitialization*. Observe that initialization can make the old value useless or unpinned.

(3) (deallocation) A variable becomes useless and its space relinquished when we assign Ω to it, e.g., by executing Q:=Ω, when Q is a set or map. When Q is a map, both its domain and range are deallocated. We can also deallocate the image of a specific domain point x under map Q by executing $Q(x)$:=Ω. Deletion operations, to be discussed, can also trigger deallocation.

(4) (addition) Adding an element x to a set S is denoted by S **with**:= x. We require two semantic conditions: $x \notin S$ and S should not be pinned. Also, if S is a variable, it should be a distinct identifier from x. In the case of maps f, the operation $f\{y\}$ **with**:=x adds y to the domain of f if it is not initially there, and adds x to the image of y under f. We require that x not belong to the image set $f\{y\}$ and that both f and $f\{y\}$ be unpinned. We also require that terms y and x must not be the same as identifier f.

(5) (link) If S is a set variable or an image set expression and T is either a set, the domain of a map, or an image set expression, then the operation $S\cup$:=T stores the value of $S \cup T$ in S and destroys the old value of S and T. We require that both S and T be disjoint and unpinned. If f and g are maps, then $f\cup$:=g stores the union of f and g in f and destroys the old value of f and g and the domains of f and g (but each image set is not destroyed). Here, we require that the domains of f and g be disjoint and that f, g, and the domains of f and g be unpinned.

(6) (copy) If S is a set variable and T is either a set, the domain of a map, or an image set expression, then the operation S:=T assigns the value of T to S. If f and g are maps, then f:=g assigns the value of g to f. As a separate case, the copy $f\{x\}$:= T requires that f is unpinned.

(7) (access) Membership tests, $x \in S$, $x \in$ **domain** f, and $x \in f\{y\}$ access sets, map domains, and map ranges, respectively. The domain of a map f is also accessed by function and multivalued map retrievals $f(x)$ and $f\{x\}$. Such operations may also appear on the left of an assignment, in which case they are indexed storage operations. Element deletions S **less**:= x and $f\{x\}$ **less**:= y are also access operations that delete elements from a set and a map range respectively. We require the conditions $x \in S$ and $y \in f\{x\}$, respectively. Furthermore, in the case of map range deletion, if we delete the last element from the image set $f\{x\}$, then x is automatically deleted from the domain of f. The deallocation operation $f(x)$:= Ω also deletes the element x from the domain of f. It is useful to distinguish between two kinds of access operations - *associative access* and *self-access*. A self-access is an operation that accesses a set Q with an argument x that is always preceded by a retrieval of x from Q. Anything else is an associative access.

4.2. Simulation of ASM by APM

We introduce in this section three kinds of data structures that support the real-time simulation of ASM by APM.

4.2.1. Unbased representations

The simplest of our data structures for implementing sets are doubly-linked lists with pointers to the first and last list cell. Each list cell stores an element of the set. For maps f, we store the domain of f in the list. For each list cell with domain value x we also store the range value $f(x)$ if f is single-valued at x or the image set $f\{x\}$ if it is multi-valued. To avoid ambiguities, we indicate whether a function is single-valued or multi-valued for each domain point.

This representation, which is called *unbased*, supports the real-time simulation of all seven of the primitive operations above except for access, where only self-access operations can be simulated in the given time bound. The problem with associative access can be illustrated with the following simple example:

(**while** $\exists x \in S$)
    ...
    Q **less**:= x
    ...
end

In the preceding code, if S and Q are both implemented as doubly linked lists, then retrieving an arbitrary value from S and storing it in x can be done in $O(1)$ time. However, the subsequent search needed to locate this value within Q in order to delete it cannot in general be achieved in unit time.

4.2.2. Based representations - Associative Access Removal

In order to solve the associative access problem mentioned above, we follow the approach found in [24] where values common to both sets S and Q are stored in one place. Consequently the unit time retrieval from S locates the value within Q as well. More generally, we introduce a universal set B called the *base* for S and Q and maintain the invariant

$S \cup Q \subseteq B$

To maintain this invariant we represent B and Q as a set of records, each record containing a B and a Q field. The elements of B are stored in the B field and serve as the key. The Q field stores Ω if the element in the B field does not belong to Q. Those records whose B field values belong to Q are connected by a doubly linked list stored within their Q fields. The first and last records of the Q field list are indicated by *first* and *last* pointers. Set S is represented as a doubly linked list of pointers to records whose B fields store the elements of S. We also use *first* and *last* pointers for the S list.

We say that the elements of S are *weakly based* and the elements of Q are *strongly based* on B. Also, an object x is weakly based if the invariant $x \in B$ is maintained and x is represented as a pointer to a record whose B field stores its value. Objects aggregated around the same base are said to be *compatible*. In general we can have an arbitrary number of weakly based objects and sets whose elements are weakly based on the same base. However, since records must have a constant (data independent) number of fields, there can only be a constant number of sets whose elements are strongly based and mutually compatible.

Observe that if each record field for strongly based sets is named, then we can always omit a field instead of storing Ω there. Wiederhold refers to this as a *pile* representation [37]. Consequently, space utilization for weakly and strongly based representations preserves the ASM space complexity.

Weakly based and strongly based representations can be used for elements of set valued variables and domains of maps. Weakly based representations can also be used for elements of nonsimple sets. However, the elements of arbitrarily overlapping nonsimple sets are difficult to represent using a strongly based data structure without sacrificing space. An exceptional case is when such sets are pairwise disjoint (e.g., the image sets $E\{x\}$ of a one-to-many mapping E), in which case an associative access uniquely determines the set to which the search argument belongs.

Let us consider how these two representations support real-time simulation for our seven basic primitive operations. Clearly, retrieval operations have the same performance as for the case of unbased sets. However, if a weakly based value x is retrieved from a set S, then the elements of S and x must be compatible. Weakly based sets support initialization and deallocation in the same way as unbased sets. However, since a single field cannot be removed in unit time from a whole set of records, strongly based sets must be undefined before they are initialized (i.e., they cannot be reinitialized), and they must be empty before they are deallocated. When adding a value x to a set S whose elements are weakly or strongly based on a base B, we require that either x is compatible with the elements of S or that $x \notin B$. In the latter case addition is implemented by adding a new base record with B field x before updating the representation of S. We allow linking two sets whose elements are compatible if the elements of both sets are based in the same way (both weakly based or both strongly based). We can also link two sets whose elements are incompatible if they are both weakly based or both strongly based and their bases are disjoint. In this case in addition to linking we must also merge the two bases (which takes unit time). Copy operations are supported for based sets, although any triggered deallocation operations must obey the simulation conditions. Since weakly based sets suffer from the same shortcoming as unbased sets, only self-accesses are allowed. However, we can perform unit time associative accesses on a strongly based set S using search argument x so long as x is compatible with the elements of S.

It should also be mentioned that any of the operations that can unpin an object (recall that these include initialization, deallocation, deletion) can also render objects useless and amenable to deallocation. When a record belonging to a strongly based aggregation has only empty non-base fields and is unpinned (in the sense that there are no live pointers to it), then the base element can be deleted and the record deallocated.

Given an ASM program that satisfies the initial semantic conditions, we know that real-time simulation is possible if

(i) each set undergoing associative access can be strongly based; and

(ii) each of the objects that must be compatible with the elements of these strongly based sets can be weakly based.

The bases are the equivalence classes of compatible objects. Any object that is not based, is unbased.

4.2.3. Refinements

The preceding data structures are overly general in order to support real-time simulation in a wide range of contexts. However, in particular situations these structures can be simplified considerably to obtain better performance.

It is worth mentioning several transformations that prune these data structures of unnecessary pointers. When lists only need to be traversed in one direction, convert a two-way list to a one-way list. If a list does not participate in any traversal, then its pointer fields can be eliminated entirely. *First* and *Last* pointers can be eliminated when they are not needed. Exogenous data structures (that store pointers through which data is accessed) can often be turned into endogenous data structures (that store data internally), whenever the data being stored is of fixed length.

Various other refinements can be made to the base representations to improve the APM simulation. When no base element ever contributes either directly or indirectly to the program output, the base set is useless and the base field of each record in a strongly based aggregation can be eliminated. In this case the address of the record can serve as the key. Whenever based sets are disjoint, they can sometimes be combined in a single field of a strongly based representation (as is done in Knuth's topological sorting algorithm [18], which is explained in [23]).

4.2.4. Subbases

In the preceding discussion we mentioned the potential space blowup when handling associative access for sets that belong to collections. We mentioned one favorable, exceptional case when the collection of nonsimple sets are pairwise disjoint. Another solution sometimes arises when each set belonging to the collection has a unique index, and when the search argument involved in the associative access is retrieved from a set that also belongs to a collection and has the same index.

For example, suppose we retrieve an element y from the image set $e\{x\}$ and then perform the access $cost\{x\}(y)$ on the domain of the function $cost\{x\}$. (Such a code pattern often occurs in graph algorithms where each element $y \in e\{x\}$ adjacent to vertex x is examined and each edge weight $cost(x,y)$ is subsequently determined.) Since both collections of sets are indexed in the same way using x, we can use an indexed base, called a *subbase*, $B(x)$ for each index $x \in$ (**domain** $e \cup$ **domain** $cost$), where invariant $e\{x\} \cup$ **domain** $cost\{x\} \subseteq B(x)$ is maintained for all x. Each set $e\{x\}$ is weakly based on $B(x)$ and each set **domain** $cost\{x\}$ is strongly based on $B(x)$. Finally, it is possible that each base set $B(x)$ could be weakly based on some other base. This example leads in a natural way to a general procedure.

4.2.5. Base set construction

Examination of many programs that benefit from basings indicates that in many cases the bases would not be modified after they are constructed. For example, in many graph algorithms, the set of all nodes of the graph often plays the role of a base set, and is either input explicitly, or implicitly as the domain of definition of the successor (edge) map. This restricted case of static bases was investigated in [24] for the final data structure selection phase of a very high level problem specification compiler. In the present paper we consider the broader context of dynamic bases that can be first constructed externally and input, and then augmented or diminished at run-time by addition, link, and deallocation operations.

We will assume that the initial based data structures are input and not constructed by the program. That is, for each based input set S with base B, the input operation for S reads both B and the based representation for S. It is assumed (an assumption common to almost all algorithms texts; cf. [2]) that the input is preconditioned in this way and the program takes no measures to ensure that this is so.

Of course, base representations could also be built by the program when data is input. This is the technique described in [27], where hashing is used to ensure that a base contains no repeated elements.

4.3. ASM simulation with ARAM

The increased power of ARAM over APM can be exploited to provide for improved implementations when simulating ASM with ARAM. For example, when sets with compatible, strongly based elements on a base B are all fairly dense relative to B, then the number of such sets could be made to depend on the input, e.g., adjacency matrices for nearly complete graphs have this property. This is because the records representing strongly based sets can be implemented as exogenous arrays with unit time cursor access. Denseness guarantees no asymptotic blowup in space utilization (as would happen with nondense adjacency matrixes).

Also, any static set or dynamic set of bounded size can be represented by an array with unit time initialization and reinitialization. Of particular importance, when a base set is either static or of bounded size, the set of records representing strongly based sets can be implemented in what Wiederhold calls *transposed* form [37]. That is, The base B is an array of, say, n values and each strongly based set S is implemented as a single array of size n. If $x \in B$ and is located in the i-th component of the B array, then the i-th component of the S array indicates whether x belongs to S or not. For this data structure, access to an element of S requires a pointer to the beginning of the B array, a cursor to the position where x is located in the array, and a pointer to the beginning of the S array. Unlike the APM, an ARAM allows sets with strongly based elements to be deallocated, initialized, or reinitialized as empty sets at run-time in unit time. However, bases implemented with arrays are only static (i.e., remain constant after they are input).

4.3.1. Refinements

Several refinements can be made to the general ARAM implementation of based data structures mentioned above. Whenever possible we can attempt to convert list and pointer operations into more efficient array and tuple operations. A bounded sized list can be turned into an array of records (to support $O(1)$ time initialization), each with three fields - one for the data, one for a cursor, and another for a stack. Of course, when efficient initialization does not improve the asymptotic complexity, then these auxiliary fields for cursor and stack can be eliminated. Pointers into the middle of an array or tuple A can be turned into cursors relative to the base address of A. An array can be used to implement one bounded stack, two disjoint bounded stacks, or one bounded queue. When initialization is not needed or not costly, a strongly based set can be implemented as a single column without a stack or cursor array. Furthermore, when a strongly based set is also not shared by more than one object, the transposed representation can be replaced by a simple record implementation. Finally, one or two bounded stacks or one bounded queue can be implemented by an array.

4.4. ARAM simulation in APM

It is interesting to consider when direct access using ARAM arrays can be avoided without penalty; that is, for example, when an ARAM program can be simulated with an APM program in real time. We consider a solution with two cases. The first and easiest is when an array or tuple *A* is always accessed at one or two fixed components or in a sequential manner. In this case we can implement *A* as a list with *next* and *prev* pointers for each dimension and two pointers *first* and *last* for entry into the list.

However, when arrays and tuples are processed in a wholely nonsequential fashion, simulation is more difficult. Fortunately, this case reduces to the method for simulating ASM with APM. To see this, just consider tuples and arrays as maps. The domains of these maps are sets of indexes, and the ranges include the component values. Using this kind of simulation, it is easy to rewrite a whole bunch of algorithms that seem to require cursor access (e.g., Knuth, Morris, Pratt fast string matching [19] or lexicographic sorting; see, [2]) in a list processing form without any asymptotic penalties in time or space.

5. Conclusion

We have outlined a new programming methodology for specifying complex data structures using a finite set theoretic language and for implementating these abstract specifications using simpler lower level storage structures. The translation from abstract to concrete data structures simulates a set machine on a RAM in real-time (where only the complexity of data access is considered). Our ideas derive from several important sources mentioned below.

Jay Earley [10] considered the problem of abstract data structure specification and implementation principles to support compilation of his proposed language VERS2. He used a graph theoretic access path model that handled both cursor and pointer access to data. However, his concrete storage structures had limited dynamic capabilities, and his design principles did not incorporate formal complexity.

After Earley, researchers in the programming language and artificial intelligence communities proposed or implemented a number of interesting expert systems for data structure design [3, 15, 21, 26] . These approaches sacrificed Earley's goal of a unified theory of data structure design in favor of hand crafted systems that could work well in a limited domain.

Schwartz and others [27, 29, 30] modelled complex data structures in SETL and provided a language (that introduced the notion of a base and based representations) for implementing finite sets, maps, and tuples using concrete storage representations. An automatic data structure selection algorithm was also implemented within the SETL optimizer. Two drawbacks with this work are that

(i) the data structures are specially designed for automatic SETL optimization and are too limited for use in a manually guided general purpose programming methodology; and

(ii) the SETL data structure design principles incorporate only an expected notion of complexity, in which hashing is assumed to take unit time.

Hoare [13] used the general notion of simulation to verify the implementation of abstract data types using concrete data types. We regard this work and other work in classical verification as revealing the inverse relationship that verification has with top down design. In this paper Hoare did not verify any aspect of complexity.

We should mention certain limitations and areas for improvement in our approach. In the ARAM model tuples are only partially dynamic in order to ensure that garbage collection would not increase the asymptotic time. It would be interesting to consider the smallest space bounds in which the use of garbage collection (instead of having infinite memory) only increases time by a constant factor when sparsely utilized tuples can be allocated and deallocated dynamically. Restricting data structure selection to a real-time simulation is probably too restrictive. It would be interesting to consider a more general simulation in which each unit time set operation is implemented in unit amortized time [35] on a RAM. Finally, some of the simulation conditions that require data to be unpinned might be relaxed if persistant data structures [9] were used.

There are several areas where our methodology might be usefully applied. One of these is in automated software development. We are investigating a data structure selection transformation based on this methodology for the RAPTS system [24]. In the area of verification, we might be able to get shorter more convincing transformational proofs by using inference rules more naturally based on our data structure design principles. A third application is pedagogical. We believe that a significant compression would be obtained by reformulating the first seven chapters of Aho, Hopcroft, and Ullman [1] using the data structure design principles outlined here. Finally, our ideas on simulating cursor access with pointer access in real time might lead to a way in which methods to make linked data structures persistent [9] could extend to cursor based data structures.

References

1. Aho, A., Hopcroft, J., and Ullman, J., *Data Structures and Algorithms,* Addison-Wesley, 1983.
2. Aho, A., Hopcroft, J., and Ullman, J., *Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.
3. Barstow, D., *Knowledge-Based Program Construction*, Elsevier North-Holland, 1979.
4. Boas, P. v. Emde, ''Preserving order in a forest in less than logarithmic time and linear space,'' *IPL*, vol. 6, pp. 80-82, 1977.
5. Boas, P. v. Emde, Kaas, R., and Zijlstra, E., ''Design and implementation of an efficient priority queue,'' *Math. Sys. Theory*, vol. 10, pp. 99-127, 1977.
6. Deak, E., ''Source to source transformation in SETL: a case study,'' *IEEE Transactions on Software Engineering*, Jan., 1981.
7. Dewar, R., Grand, A., Liu S. C., Schwartz, J. T., and Schonberg, E., ''Program by Refinement as Exemplified by the SETL Representation Sublanguage,'' *TOPLAS*, vol. 1, no. 1, pp. 27-49, July, 1979.
8. Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.
9. Driscoll, J., Sarnak, N., Sleator, D., and Tarjan, R., ''Making Data Structures Persistent,'' in *Proc. 8th ACM STOC*, pp. 109 - 121, ACM, May, 1986.
10. Earley, J., ''Toward an Understanding of Data Structures,'' *CACM*, vol. 14, no. 10, pp. 617-627, Oct. 1971.
11. Freudenberger, S., Schwartz, J. T., and Sharir, M., ''Experience with the SETL Optimizer,'' *ACM TOPLAS*, vol. 5, no. 1, pp. 26-45, 1983.
12. Hecht, M. and Ullman, J. D., ''A simple algorithm for global data flow analysis programs,'' *SIAM J. Computing*, vol. 4, pp. 519-532, 1975.

-- --

13. Hoare, C. A. R., ‘‘Proof of Correctness of Data Representations,’’ *Acta Informatica*, vol. 1, no. 19, pp. 271-281, 1972.

14. Kam, J. B. and Ullman, J. D., ‘‘Global data flow analysis and iterative algorithms,’’ *JACM*, vol. 23, no. 1, pp. 158-171, 1976.

15. Kant, E., *Efficiency in Program Synthesis,* UMI Research Press, 1981.

16. Kennedy, K., ‘‘Node Listings Applied to Data Flow Analysis,’’ in *Proceedings 2nd ACM Symposium on Principles of Programming Languages,* Jan, 1975.

17. Kildall, G., ‘‘A Unified Approach to Global Program Optimization,’’ in *Proc. First ACM Symp. on Principles of Programming Languages*, 1973.

18. Knuth, D. E., *The Art of Computer Programming, Vol 1: Fundamental Algorithms,* Addison-Wesley, 1973.

19. Knuth, D., Morris, J., and Pratt, V., ‘‘Fast Pattern Matching in Strings,’’ *SIAM J. Computing*, vol. 6, no. 2, pp. 323-350, 1977.

20. Kruchten, K., Schonberg, E., and Schwartz, J., ‘‘Software Prototyping using the SETL Programming Language,’’ *IEEE Software*, pp. 66-75, Oct., 1984.

21. Low, J. and Rovner, P., ‘‘Techniques for the automatic selection of data structures,’’ in *Third POPL,* pp. 58 - 67, 1976.

22. Mehlhorn, K., *Sorting and Searching,* Data Structures and Algorithms, 1, Springer-Verlag, 1984.

23. Paige, R., ‘‘Transformational Programming -- Applications to Algorithms and Systems,’’ in *Proceedings Tenth ACM Symposium on Principles of Programming Languages*, pp. 73-87, Jan, 1983.

24. Paige, R. and Henglein, F., ‘‘Mechanical Translation of Set Theoretic Problem Specifications Into Efficient RAM Code - A Case Study,’’ *Journal of Symbolic Computation*, vol. 4, no. 2, pp. 207-232, Aug., 1987.

25. Paige, R and Tarjan, R., ‘‘Three Efficient Algorithms Based on Partition Refinement,’’ *SIAM Journal on Computing*, vol. 16, no. 6, Dec., 1987.

26. Rovner, P., ‘‘Automatic Representation Selection for Associative Data Structures,’’ TR10, Dept. of Computer Science, University of Rochester, 1977. Ph. D. Thesis, Harvard University

27. Schonberg, E., Schwartz, J. T., and Sharir, M., ‘‘An Automatic Technique for Selection of Data Representations in in SETL Programs,’’ *ACM TOPLAS*, vol. 3, no. 2, pp. 126-143, Apr, 1981.

28. Schwartz, J., Dewar, R., Dubinsky, D., and Schonberg, E., *Programming with Sets: An introduction to SETL,* Springer-Verlag, 1986.

29. Schwartz, J.T., ‘‘Automatic Data Structure Choice in a Language of Very High Level,’’ *CACM*, vol. 18, no. 12, pp. 722-728, Dec, 1975.

30. Schwartz, J. T., ‘‘Optimization of Very High Level Languages, Parts I, II,’’ *J. of Computer Languages*, vol. 1, no. 2,3, pp. 161-218, 1975.

31. Suppes, P., *Axiomatic Set Theory,* Dover, 1972.

32. Tarjan, R., ‘‘A Class Of Algorithms Which Require Nonlinear Time To Maintain Disjoint Sets,’’ *J. Comput. Sys. Sci.*, vol. 18, pp. 110-127, 1979.

33. Tarjan, R., *Data Structures and Network Algorithms,* SIAM, 1984.

34. Tarjan, R., ‘‘Iterative Algorithms for Global Flow Analysis,’’ in *Algorithms and Complexity, New Directions and Recent Results,* ed. Traub, J., pp. 91-102, Academic Press, 1976.

35. Tarjan, R., *Amortized computational complexity*, 6, pp. 306-318, SIAM J. Alg. Disc. Meth., 1985.

36. Tenenbaum, A., *Type Determination for Very High Level Languages*, New York University, Dept. of Computer Science, Oct, 1974. appears in Courant Computer Science Report 3

37. Wiederhold, G., *Database Design*, McGraw-Hill, 1983.