Stanford Artificial Intelligence Laboratory
Memo AIM-306

November 1977

Computer Science Department
Report No. STAN-CS-77-639

ON PROGRAM SYNTHESIS KNOWLEDGE

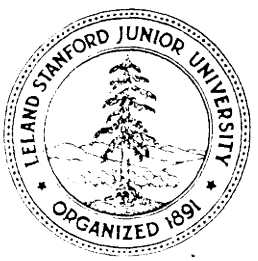
by

Cordell Green
David Barstow

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University



Stanford Artificial Intelligence Laboratory
Memo AIM-306

November 1977

Computer Science Department
Report No. STAN-W-77-639

ON PROGRAM SYNTHESIS KNOWLEDGE

by

Cordell *Green*
David **Barstow**

ABSTRACT

This paper presents a body of program synthesis knowledge dealing with array operations, space reutilization, the divide and conquer paradigm, conversion from recursive paradigms to iterative paradigms, and ordered set enumerations. Such knowledge can be used for the synthesis of efficient and in-place sorts including quicksort, mergesort, sinking sort, and bubble sort, as well as other ordered set operations such as set union, element removal, and element addition. The knowledge is explicated to a level of detail such that it is possible to codify this knowledge as a set of program synthesis rules for use by a computer-based synthesis system. The use and content of this set of programming rules is illustrated herein by the methodical synthesis of bubble sort, sinking sort, quicksort, and mergesort.

David Barstow is currently in the Computer Science Department, Yale University, 10 Hillhouse Ave., New Haven, Conn 06520.

This research was supported in part by the Advanced Research Projects Agency under ARPA Order 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University or the U.S. Government.

ON PROGRAM SYNTHESIS KNOWLEDGE

Cordell Green
David Barstow

MARCH 1977

Keywords: program synthesis, programming knowledge, automatic programming, artificial intelligence

This paper presents a body of program synthesis knowledge dealing with array operations, space reutilization, the divide and conquer paradigm, conversion from recursive paradigms to iterative paradigms, and ordered set enumerations. Such knowledge can be used for the synthesis of efficient and in-place sorts including quicksort, mergesort, sinking sort, and bubble sort, as well as other ordered set operations such as set union, element removal, and element addition. The knowledge is explicated to a level of detail such that it is possible, to codify this knowledge as a set of program synthesis rules for use by a computer-based synthesis system. The use and content of this set of programming rules is illustrated herein by the methodical synthesis of bubble sort, sinking sort, quicksort, and mergesort,

David *Barstow* *is currently in the Computer Science Department, Yale University, 1 0 Hillhouse Ave., New Haven, Conn 06520.*

This research was supported in part by the Advanced Research Projects Agency under ARPA Order 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University or the U.S. Government.

1. Introduction

In this paper, we present a body of program synthesis knowledge and suggest that a computer can be programmed to use this knowledge to write several very good sort programs such as quicksort and mergesort as well as related programs such as set union, and basic enumeration operations.

This paper is an extension of two earlier papers [8, 9], parts one and two of this series *OH* our theory *of programming,* or a codification of programming knowledge. The first paper presented rules as English statements, and the second showed how these rules could be translated into computer usable form. The earlier papers discussed the rules for iterative transfer-paradigm sorting programs -- selection and insertion sorts. This paper presents programming knowledge, again in English, about the divide-and-conquer paradigm, space re-utilization, and other ordered set operations. This knowledge allows the synthesis of the previously synthesized sorts but also allows more naturally recursive sorts such as quicksort and mergesort. Another addition in this paper is the discussion of arrays as data types, which combined with space re-utilization knowledge allows the synthesis of in-place insertion, Selection, sinking, and bubble sorts, as well as quicksort, and merge sort.

The reader may agree that this programming knowledge is relevant for the synthesis of sorting algorithms but wonder what generality it possesses. We believe that the programming knowledge presented in this paper, combined with that of our earlier two papers, extends well beyond sorting. For example, the reader will observe that the sorting programs discussed herein include simple searching, ordered set union, removal of an element from a set, additions of an element to a set, etc. Much of what occurs in programs consists of set enumeration operations in the form of loops or recursions on arrays or lists. Thus our

low-level programming rules for operations on lists and arrays and for space reutilization should be widely applicable. The higher-level transfer paradigm, recursive paradigm, and divide-and-conquer paradigm will be used less frequently than the low level rules, but are certainly applicable outside the sorting context. Knuth [11] has stated that "virtually every important aspect of programming arises somewhere in the context of sorting or searching," so we feel sorting is a good starting point.

There is also now some empirical evidence suggesting that this approach may have wider utility. We have expressed a subset of these rules in a programming formalism [9]. These were used in a rule-testing system that included about 150 rules and produced a variety of sort programs. Further rules were tested in a larger program synthesis system [2, 1]. The synthesis system used these rules and further refinements to synthesize simple learning programs, linear prime finding algorithms, reachability algorithms and information storage and retrieval programs. The basic rules showed considerable carry over to new applicatons. The issue of generality is very complex, depending upon the class of application programs synthesized and the larger context in which the synthesis rules are used, so that it is much too early to reach any conclusions about this approach except that it shows promise.

We neither claim nor believe that the particular synthesis rules and paradigms expressed here are "optimal" in any sense. Instead, we hope we have provided a starting point and that other researchers will introduce better rules and further refinements of those presented here. We especially wish to make no claims for the *synthesis paths* or particular derivations given. The particular synthesis paths we gave were chosen to better explicate the programming rules, but it -is likely that a synthesis system would make choices in different orders in addition to following totally different synthesis paths.

For the reader who has missed our earlier papers, we summarize in appendix 1 a subset of the earlier rules, the enumerator rules, and indicate how they may be programmed. These rules cover enumeration of elements and positions in stored sets, according to given ordering relations, They cover lists and arrays as data structures and include enumerator constructs such as initialization, termination tests, and methods of saving or marking the state of an enumeration as it progresses. In this paper we will not descend to the detailed level such as assignments and low-level list and array operations, as these were covered earlier.

Our method of describing programming knowledge can be viewed as detailed stepwise refinement or, alternatively, logical program synthesis by special rules of inference. One comment we have received is that it is a programming theory that could be taught to students so that they can synthesize programs by learning these rules and paradigms. Perhaps it is a good theory for humans to use, but it has been designed for computers to use, and we've not tested it as a teaching method. We present our rules in English, rather than a mathematical, logical, or procedural formalism in this paper in order to make the knowledge easier to follow. The refinement rules and the programming concepts may also be viewed as a planning spece that structures and reduces the search to a more orderly generation of reasonable programs, rather than a generation of all possible (frequently meaningless) programs. In the heuristic search paradigm, the refinement rules may be thought of as "plausible move generators" whose goal is to generate only programs that should be considered in a given context.

Although we will primarily follow the stepwise refinement paradigm, we also invoke a hypothesize and test method, and an inferential capability for simplifying tests and producing speed-ups. The use of these methods allows more sophisticated syntheses, but makes the subsequent transformation into rules more complex and less deterministic. The refinement

methods discussed include some simple, heuristic rules to motivate our choices, In fact our rule testing program is designed to interact with an "efficiency analysis program" that decides which of the alternatives are more efficient. The efficiency estimation also prunes the search for reasonable programs. We are not claiming any great precision in these efficiency rules, just that they may, often be used as a guideline. Any accurate determination of efficiency will take more detailed calculations. A more rigorous discussion of how to automate the efficiency analysis is given by Kant [10]. A dsicussion of the close iterrelation between refinement and efficiency considerations is given by Barstow and Kant [2]. Thus two techniques are provided for reducing the search for programs -- a refinement style planning space and an efficiency estimator. We feel that these will still not be adequate and that additional heuristics will be required to factor subprograms into independent or near-independent modules.

-Much of the program synthesis knowledge and the general approach presented here are embodied in a program which is the "coder" and are also embodied to some extent in the "efficiency expert" of the much larger PSI program synthesis system [6,7]. The PSI system consists of two phases: an acquisition phase and a synthesis phase. The acquisition phase constructs a -high-level model of the desired program and information structures from a dialogue with the user. The dialogue contains English as well as examples and traces. The acquisition phase consists of a parser-interpreter written by Jerrold Ginsparg [5], a trace and examples inference system by Jorge Phillips [17], a dialogue moderator by Lou Steinberg, a domain expert by Ronny Van den Heuvel, and a model builder by Brian McCune [15].In the synthesis phase the coder and efficiency expert interact to refine the high level program model into executable and efficient code. The synthesis phase consists of the coder by David Barstow and the efficiency expert by Elaine Kant.

The most closely related **work to that presented here is** that of John Darlington **[4].** He also presents a formalism for the synthesis of essentiaily the same class of sorting programs. His formalism is quite different, and **is based upon algebraic** characterization of program transformations. Darlington and Burstall **[3]** present other techniques for **recursive-to-**iterative program **transformations.** Manna and Waldinger **[14]** and Laaser **[12]** investigate alternative techniques for synthesizing programs for finding the extrema of a set by finding recursive programs through problem reduction to equivalent subproblems applied to smaller sets,

2. Overview of Programming Knowledge

To help provide **an** overview, we present below a summary of some of the key program synthesis constructs used in this paper. These will be elaborated through the vehicle of tracing the synthesis of the various sort programs.

Divide and conquer paradigm
- **correctness conditions**
- **uniform recursive**
- **choice of partitioning** *method*
  - **singleton, equal-size**

Transfer paradigm
- **recursive to iterative transformation**
  - **hypothesize and test method**
  - **sufficient conditions**

In-place operations
- **feasibility**
- **re-use of "no** longer referenced" sets, locations
- **in-place element addition and deletion**
- **shifts, ordered and unordered**

Sets stored in contiguous regions of arrays
- **fixed and movable boundaries**
- **minimal shifting for insertions and deletions**
- **positions for placement of sets**

Simplifications of insertions, deletions, position and element testing

Enumerate and process
- **positions, elements**
- **selection of enumeration orders**
  - **left-right, right-left, binary chop, largest first, alternating**

Enumeration simplifications
- **early stop, late start**
- **use of transitivity, re-use of earlier comparisons**
- **enumeration merging**
  - **compares and** shifts
- **compare and shift by exchanging**
- **finding best by candidate replacement**

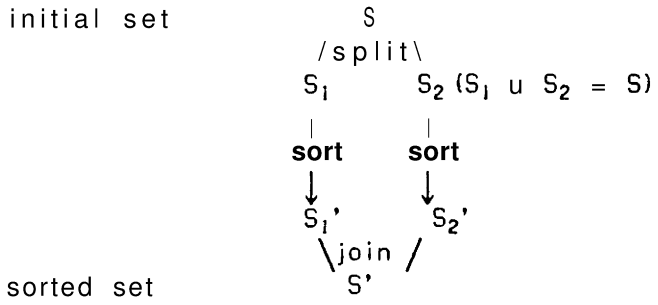
Ordered set union
- **insertion union,** selection union

PROGRAM SYNTHESIS CONSTRUCTS USED

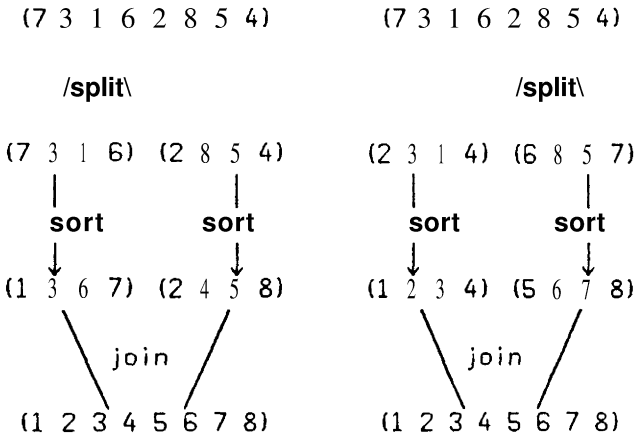
3. The Divide-and-Conquer, or Partitioning Paradigm

The top-level technique we shall consider is to split the set to be sorted (the input) into parts, sort the parts, and then put the parts back together to form the output. The three operations split, sort and join must be done cooperatively in such **a** manner that the whole set is sorted. We see that this method is naturally recursive, for we may use the same sorting algorithm to sort each of the parts. This general paradigm **has** several names **including** recursive sorts, partitioning sorts or divide and conquer sorts.

The set to be sorted may be divided into **a** number of parts. We shall assume in this paper that the set will be divided into two parts. The method may be diagrammed as shown. The set S is split into two disjoint sets S_1 and S_2. These are sorted Into S_1' and S_2', which are then joined to form the sorted set S'.



Two instances of this paradigm would be mergesort and quicksort, as illustrated below:



(using 4 as the dividing element)

Note that in mergesort, the set is split into a left and right half, according to the position of the elements. The join or merge operation does some sorting-like work to merge the two parts. Note that in quicksort the split does more work, dividing the set into ail elements less than or equal to 4, and all elements greater than 4. The join operation is simpler, being just an append. In both cases the split is intended to produce sets of approximately equal size. Mergesort is somewhat analogous to insertion sort and quicksort is somewhat analogous to selection sort. For insertion and selection sorts, the split produces one subset with one element and another subset with all the rest. The analogy carries over to produce the taxonomy summarized in the following diagram:

	SINGLETON SPLIT	EQUAL SIZE SPLIT
WORK DONE BY JOIN	INSERTION (OR SINKING)	MERGESORT
WORK DONE BY SPLIT	SELECTION (OR BUBBLE)	QUICKSORT

A TAXONOMY OF SOME SORT PROGRAMS

In. our paradigm, exchange sorts such as sinking or bubble sort are seen as minor variations on other forms of sorts.

One may state precise mathematical conditions that are sufficient for this partition, sort, and join technique to work. For the purposes of this discussion, we will assume that no elements are repeated. As in the case above, we let S be the initial set, and S_1 and S_2 be the subsets obtained by the split operations, $split_1(S)$ and $split_2(S)$. Thus the final 'set S' is the set $join(sort(split_1(S)), sort(split_2(S)))$. Let the predicate ORDERED mean that the order in which the sets are stored corresponds to the ordering which is to be achieved by the sorting operation, We will use "=" to mean that the sets have the same elements. Then one way to state sufficient conditions is to state that the split, sort, and operations do not gain or lose elements inappropriately, i.e.

$S = split_1(S) \cup split_2(S)$ and $join(S_1, S_2) = S_1 \cup S_2$

Also, the split must divide the set into smaller subsets, and the sort operation must appropriately order the elements, i.e.

$$ORDERED(sort(S_i))$$

and the join must preserve that ordering, i.e.

$$ORDERED(S_1) \text{ and } ORDERED(S_2) \Rightarrow ORDERED(join(S_1, S_2))$$

Note that we have not stated how each subset is to be sorted. Indeed, it is possible within the framework given so far, for a different algorithm to be used to sort the subsets, and so forth. If one is repeatedly performing an operation, as in the recurrent sort operations performed on the smaller subsets, and the same mechanism is used, then a *uniform* method results. For more optimal algorithms, a different method may be selected, either in advance or dynamically for subsequent occurrences of the same operation. In fact, a very good sort is to use a recursive uniform quicksort until the subsets reach a certain size, then use an insertion sort for the subsets [18]. In rule form, our first rule is as follows:

1) choose the number of partitions into which the set shall be split
2) choose a split operation
3) write a sort program for each subset
4) choose a join operation

all subject to the conditions stated above.

If the same sort mechanism is chosen for each level, then we have a recursive technique, uniform at each level, except possibly for the end cases when the sets are reduced to singleton or empty sets. For the rest of this discussion, we will be concerned only with such uniform recursive sorting techniques. For such algorithms, we may use an inductive form of the correctness specification. The inductive form states that if: (1) the empty set is sorted; and if (2) the split subsets are sorted implies that the join of the sorted Subsets is sorted; then (3) the recursive paradigm works properly, i.e. the recursive sort

$$SORT(x) \leftarrow JOIN(SORT(SPLIT_1(S)), SORT(SPLIT_2(S)))$$

(with appropriate termination test) is correct. Note again that these conditions allow different forms of split and **join, as long as together they do** the **right** thing. Obviously this technique may be extended to allow the set to be partitioned into more than two subsets.

4, Internal Representation of Sets

Before we begin the synthesis of insertion and selection sorts, let us briefly discuss the internal representation of ordered sets. Until now ail sets have existed as abstract entities, and we have not stated how they are represented in the computer. Assume that ail sets are ordered and ail elements are *explicitly represented* rather than computed by some algorithm.

Both the ordering relationship and the members of the set must be represented. Often the ordering is implicit and may be easily computed from the stored representation. For example the elements may be stored in a linked list and the traversal order of the list provides an ordering relation. Or the set may be stored in an array, with one member per array element, where the normal array ordering (first to last, say) provides an ordering on the set. Thus the set is put in a correspondence with the integers, the ith array element being ith in the ordering. This correspondence can be separate, as in the case of a separate index array the same size as the set, where the ith element holds the integer position in the ordering of the *i*th element of the original array. There are many other representations such as bit maps or tree representations. in this paper we wilt discuss algorithms that place sets in explicit order, where the "natural" storage order will hold the ordering relation.

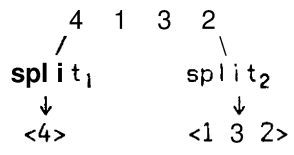
Furthermore, we will speak as if the elements are all numbers and that the final order desired is that the numbers be in increasing order. We wilt also assume that no elements are repeated. These assumptions simplify the discussion, but do not significantly affect generality.

5. Singleton Split

The first case we will consider is that in which the split operation results in one set with a single member and a second set with the remaining elements of the original set. For example, one possibility is:

$S = \langle 4\ 1\ 3\ 2 \rangle$
$split_1(S) = S_1 = \langle 4 \rangle$
$split_2(S) = S_2 = \langle 1\ 3\ 2 \rangle$

or shown graphically:



In this special case, only the second of the two subsets (S_2), must be sorted before the two are joined as the final step in the sorting operation. As mentioned above, we will consider only the case where the sorting operation for this subset is the same as the top-level sorting operation. The general paradigm may be simplified, thus

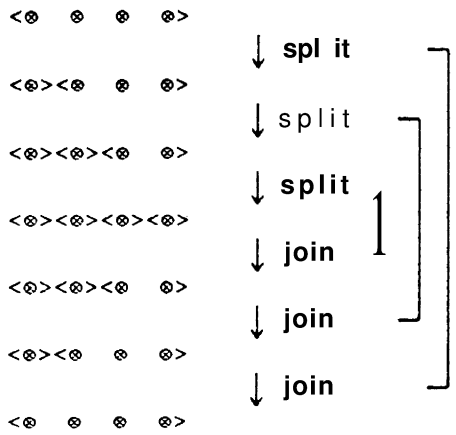
$$sort(X) \leftarrow join(sort(split_1(X)), sort(split_2(X)))$$

becomes

$$sort(X) \leftarrow join'(split_1'(X), sort(split_2(X)))$$

where join' and split,' may be further simplified to deal with single elements rather than singleton sets.

Let us first look rather closely at the operation of programs within this paradigm, as illustrated in the following diagram.



According to the above recursive formulation the split operations are all performed first, followed by the join operations. Notice also the center line of the diagram. Here we see that all of the elements of the original set have been separated from each other. In effect, the stack used by the successive calls to the sort routine is a kind of intermediate storage or buffer. Notice also that the order in which the elements are added to the buffer by the split operation is the opposite of that in which they are removed by the join operation. We will later see that the split and join operations can be interleaved, and there is no need for the stack.

The diagram also illustrates that there is a sequence of input and output sets, beginning with the full input set and empty outpt set and concluding with the full output set and the empty input set. Each split operation produces a new input set with one fewer member. For example, such a sequence might be

{ <4 1 3 2> , <1 3 2> , <3 2> , <2> , < > } ,

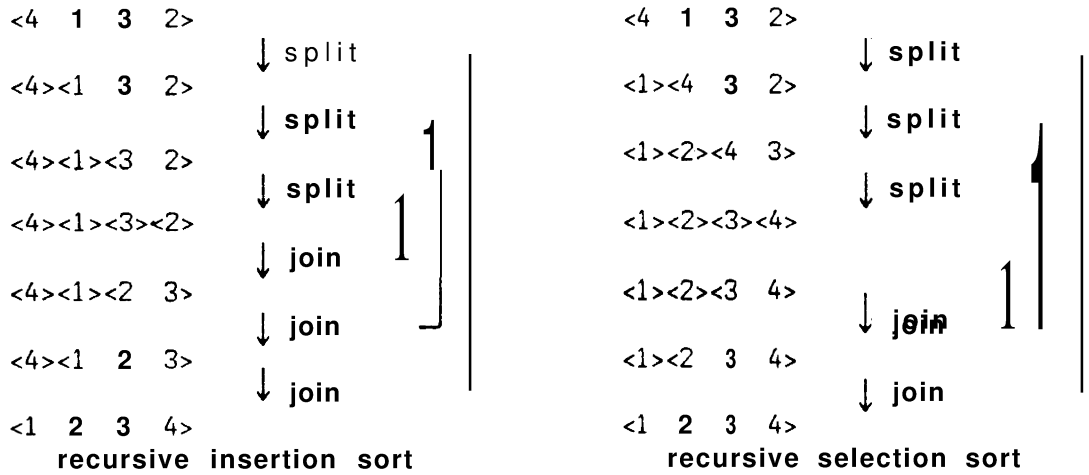
formed by- the act of removing the first element each time. This sequence may be called the input sequence of sets, although we will sometimes refer to it as the "input set", hopefully without causing confusion. Note that the output is also formed through a sequence of sets of increasing size. For example,

{ < > , <4> , <1 4> , <1 3 4> , <1 2 3 4> }

is the output sequence of sets or **just the "output set".**

5.1 insertion vs. Selection

Sort programs in which the split, always produces a singleton set can be classified into two types, depending on the nature of the split and join operations. These two types are generally referred to as *insertion* **and** *selection* sorts, and are illustrated in the following diagram



Insertion sorts are characterized by split operations which take any convenient element (e.g., the first) from the input set and join operations which do an "insertion": the new element must be added to the output set in a position which is dependent on the values of the new element and the other elements of the output set. Selection sorts are characterized -by simple or efficient join operations and split operations which do **a "selection"; i.e.** the chosen element is dependent on the values of all of the other elements.

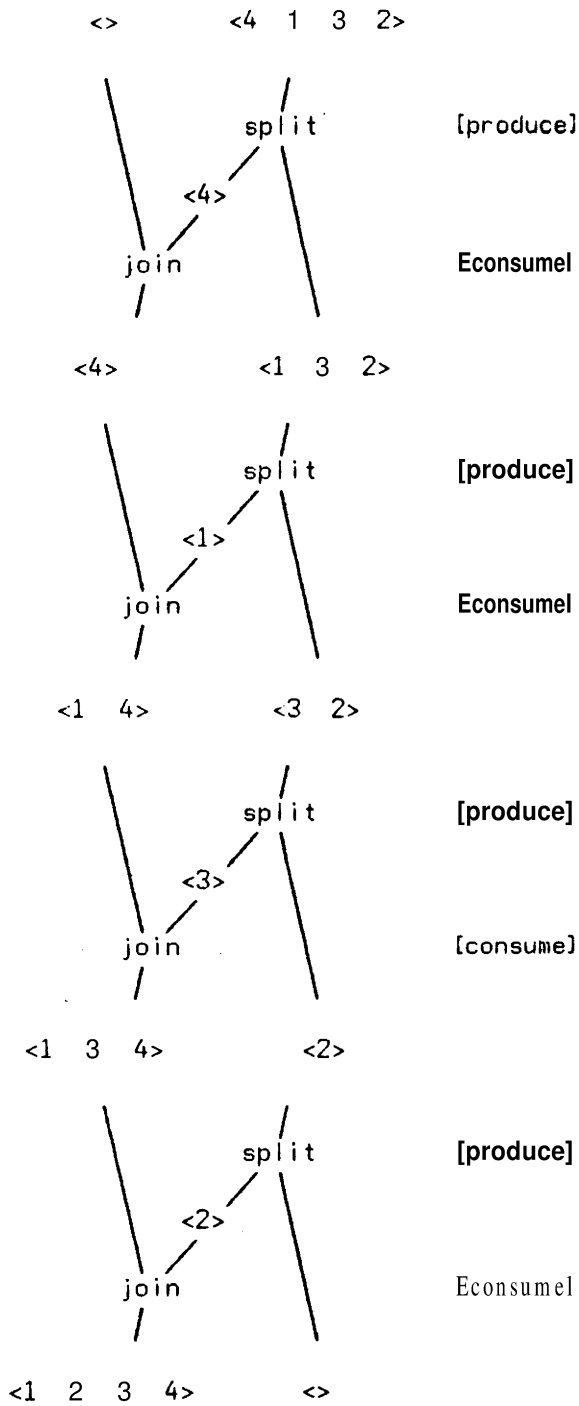
What split operations are "efficient" for an insertion sort is, of course, dependent on the particular data structures used to represent the sets. It is efficient to minimize shifting in arrays and searching in lists. This efficiency typically occurs by selecting the first or last element. The insert (join) operation will typically require searching the output for the appropriate position for the new element (e.g., a linear scan or a binary search) followed by some kind of set modification to add the element at that position (e.g., shifting array elements by one or modifying pointers in a linked list).

For selection sorts, efficient join operations usually add the new element at the front or back of the output set. The corresponding selection (split) operations take the largest element or take the smallest element. In either case, the selection operation will usually require an enumeration of the elements of the set.

6.2 Transformation from recursive produce-consume to iterative transfer paradigm

We would like to show, for selection and insertion sorts, how the recursive partitioning paradigm presented in this paper can be reduced to the "transfer paradigm" presented in our earlier papers. The transfer paradigm consists of a selection operation which takes one element from the input and puts it into some buffer, and a construction operation which takes one element from the buffer and puts it into the output set in an appropriate position. This algorithm may be implemented as two concurrent processes, a producer (selector) and a consumer (inserter) with a storage buffer between them. The singleton divide and conquer paradigm is a type of produce-consume process in which the split is a producer and the join is a consumer. The recursive algorithm can be viewed as a series of "produce" (split) operations, followed by a series of "consume" (join) operations. The buffer is the recursion

stack.. If the elements can be produced one at a time and each element in the buffer can be consumed as soon as it is produced, then the buffer size can be bounded to be of size one. In this case the produce and consume operations can be interleaved to form a sequence: produce, consume, produce, consume, etc. A straightforward implementation is a loop or iteration calling first the producer then the consumer. The interleaved sequence of operations for the insertion sort may be illustrated as follows.



The iterative selection sort is **analagous.**

-This transformation is that which is necessary to convert the recursive function:

```
sort (x) ← if empty(x) then <empty>
                       else join(split1(x), sort(split2(x)))
```

into the loop

```
sort (x) ← while ¬empty(x) do
                       y ← join' (split1'(x), y)
                       x ← split2'(x)
                  output(y)
```

note that different forms of $split_1$, $split_2$ and join may be needed in the iterative form (as indicated by $split_1'$, $split_2'$, and join').

We will use a "hypothesize and test" approach to recursive-to-iterative transformation rather than a set of program transformation rules. The transformation rule approach assumes that a set of syntactic transformations combined with some constraints is adequate to convert recursive to iterative programs. This is certainly true for simple recursion removal as is done in compilers, but in general the transformation process is quite complex, and closely interrelated with time and space efficiency issues. Furthermore, the iterative code may be considerably different from the recursive code.

We suggest that the system hypothesize the existence of components necessary for an iterative version (or a bounded buffer size version). Then the system attempts to synthesize these components but does not necessarily begin with existing recursive pieces of code. The synthesis system may answer that the components are too inefficient to pursue or too expensive to synthesize, or it may produce satisfactory code bearing little syntactic relation to the recursive code. This approach of meta-level hypothesize and test of iterative versions may also prove advantageous in the general case where the existence of an iterative version is undecidable, but one is willing to spend a certain amount of resources in the attempt to find an iterative version.

Let us exemplify this approach by suggesting a method by which a singleton-split partitioning paradigm can be reduced to an iterative transfer paradigm. This method may be formulated as constraints on the producer and consumer that are synthesized. Later in the paper we will follow the synthesis paths for these components.

First, we hypothesize the existence of a producer that can produce *the elements one* at a *time.* Obviously any singleton split operation produces the elements one at a time, and satisfies this condition. (A type of sort that does not effectively satisfy this condition is quicksort,' since the efficiency of quicksort is derived by its splitting the elements off **serveral** at a time. (See section 8.1) Next, we hypothesize the existence of a consumer that can *consume the elements one at a time.* Any join that takes a singleton and a set as its two arguments satisfies this condition. (A type of sort that does not effectively satisfy this condition is mergesort, which derives its efficiency by joining ever-larger ordered sets, using the ordered property to perform an efficient join. (See section 8.2) The production and consumption operations so derived must not depend upon the identity or order of any elements already produced but not consumed. Such a dependence would require a buffer of size larger than one to hold these elements.

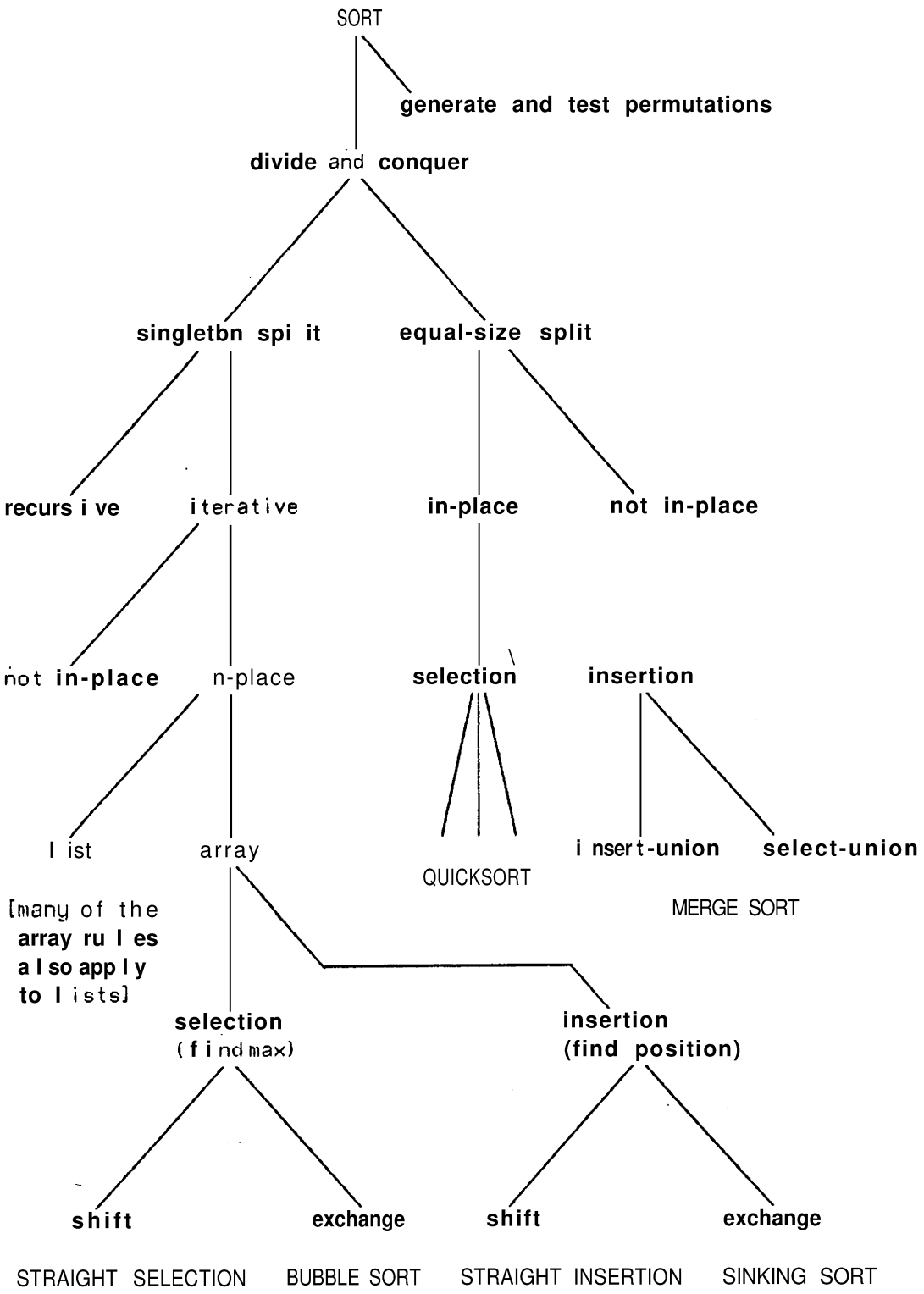
The third condition is that it must be possible to *interleave the operations* of the producer and consumer which are synthesized. That is, the elements must be consumed as soon as they are produced. For the sorting case, if the elements are consumed in the order they are produced, the induction correctness specifications must be satisfied, i.e. the output set must remain sorted after each produce-consume operation.

The insertion sort satisfies these conditions since the inserter works for any order of producing elements. For the selection sort to satisfy these conditions, it is necessary to

match the producer and consumer, e.g. a largest-to-smallest selector and a right-to-left **insertor** would work. The derivation we will give later for the selection and insertion algorithms will synthesize producers and consumers that satisfy ail these conditions necessary for the iterative transfer-paradigm.

6. Refinement tree for sorting programs

At this point, let us introduce a summary of the refinement tree we are following in this paper. Note that so far we have chosen the divide and conquer branch and the iterative branch. We are about to take the in-place branch. The reader is urged to refer back frequently to this refinement tree to avoid getting lost.


SORT
generate and test permutations
divide and conquer
singletbn spi it
equal-size split
recurs i ve
iterative
in-place
not in-place
not in-place
n-place
selection
insertion
I ist
array
i nsert-union
select-union
QUICKSORT
[many of the array ru I es a I so app I y to I ists]
MERGE SORT
selection (f i nd max)
insertion (find position)
shift
exchange
shift
exchange
STRAIGHT SELECTION
BUBBLE SORT
STRAIGHT INSERTION
SINKING SORT

7. in-place sorting

7.1 Feasibility of an In-Place sort.

With respect to the selection of data structures to represent our sets, we have assumed only that the sets wilt be stored explicitly and will be sorted into ascending order, The ascending order wilt be exhibited by the implicit ordering of the data structure (called the storage order or stored order), e.g. first-to-last for a linked list or according to increasing index for an array. We have made no assumptions about which memory locations will be used and in particular about whether memory locations can be re-used.

In synthesizing a sort algorithm it is important to find ways to conserve space at least for very large sets. For example, if in the transfer sort we store one element of each set in one memory location, and each intermediate set in the input and output sequences consumes a new set of memory locations, then it would require $O(n^2)$ memory locations to sort n elements (we will use the $O(n^2)$ notation to mean approximately or "order" n^2). If we do not require that initial and intermediate sets be "saved", i.e. if we allow them to be "destroyed" by re-using their memory locations for the newly created sets then great space savings are possible and an n-element sort can be accomplished using only O(n) memory locations. Such sorts are referred to as *in-place* sorts. We shalt now investigate in-place sorts and array representations in particular.

We-wilt consider in this section only the special case of in-place sorts where the split operation divides the set into a singleton and the rest. We wilt see that the classes of sorting programs which we have referred to as "selection" and "insertion" can lead to in-

place sorts that are often referred to as "bubble" and "sinking" sort programs [11] (pp.81 ,107).

We will now show that an in-place sort is possible. The first step is to show that the initial and intermediate sets may be destroyed, and the next is to show that only approximately n memory locations will be needed during the sorting process.

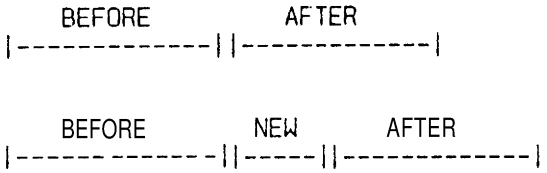
First, under what conditions is it possible to destroy the initial and intermediate sets? Let us assume that neither the initial set nor any intermediate set in the sorting process is needed after the sorting algorithm is complete. If for some reason the initial set is needed after it is sorted, then before sorting it can be saved by copying it over into a new position. For intermediate sets, after they are no longer referenced by any step of the algorithm, they can be destroyed and their memory locations can be recycled.

Consider an arbitrary set in the input sequence. The only time it is referenced is during the split operation that splits the set into one element and the rest. One can verify that this is the only reference by examining all subsets in the split-sort-join paradigm and observing that each set is named only once. Similarly, each set in the output sequence is referenced only once (when the new element is added to it). Thus the initial and ail intermediate sets may be destroyed after they are used.

Now consider the amount of computer memory that will be used to provide storage for a set having n elements. The memory required is that memory needed to indicate which are the elements of each set (a correspondence between each set and its elements) and to remember the ordering of the set. One unit of storage or one memory location is adequate per element of each set represented (in one array element or one list ceil).

Next consider the total amount of space necessary at any step of the computation, In the iterative paradigm, the algorithm consists of a sequence of transfer operations, transferring one element from a set in the input sequence to a set in the output sequence. Before each transfer operation space is required only for the current input set and tjie current output set, i.e. space for n elements. During. the transfer one space is needed to hold the element being transferred. Additionally, space will be required to store the state of enumerations of elements of the input set or of positions in the output sets, and some space could be, required for bookkeeping in the set insertion or deletion operation. The exact amount of space for each of these memory requirements is dependent on the representation chosen. The new input set requires one less space and the new output set requires one more. Thus n spaces hojd the old input and output set and n spaces hold the new input and output set. If no storage is reclaimed during the transfer, then 2n spaces plus some overhead are adequate. After the transfer the old input and output sets can be overwritten for the next transfer.

The 2n memory requirement can be reduced to n if the element addition and element deletion can be done "in-place", i.e. using all the old locations plus or minus one. Let us show that one element can be added to an ordered set stored in m contiguous iocatons to produce a new ordered set stored in mti using the original m locations plus one adjacent location plus some overhead. The conditions that must be satisfied by the transfer are that no elements are lost or gained except for the added element. Also the order must be unchanged except for the new element. Suppose that the new element NEW is larger than all elements in the subset BEFORE and smaller than ail elements in the subset AFTER. if there is a space to the right of AFTER (or to the left of BEFORE), we may add an element by shifting AFTER to the right (or BEFORE to the left) and placing the element in the vacant position as shown here.



Note that elements are conserved in BEFORE and AFTER, and the new element is added. The ordering of BEFORE and AFTER ancl all orderings between the two sets are preserved. Finally, the stored order of NEW is correct. We assumed that the shift of AFTER preserved elements and orderings. Suppose that shift is not a primitive, and we must derive a suitable shift algorithm.

The shift may be stated as a divide-and-conquer where the sort does no re-ordering. it may be reformulated more simply as a singleton-split recursive total transfer, but if it is recursive, then a buffer is used to store the elements. However, it is feasible for the shift to be reduced to an iterative transfer by the following reasoning. Elements can be produced one at a time by a total scan. Elements can be stored one at a time by any total scan of positions and by writing into array locations. The ordering is preserved if both scans are the same (i.e. left-to-right). Can the elements be consumed as soon as they are produced? **Use** the facts that only one element can be stored in one location and it is ail right to overwrite a location after the contents are no longer referenced. if a left-to-right scan is **used,** and we try to consume the first element as soon as it is produced, then the second element is overwritten. When the second element is referenced later it will not be available. (For a left-to-right scan, an "inchworm" style shift would be required, using a buffer to hold the element to be shifted next.) If a right-to-left scan is chosen then the elements can be consumed as soon as produced. An inductive argument suffices: the first element can be

moved to the right, and after the i^{th} element is moved, its place is no longer needed and the $(i+1)^{st}$ element can be moved into its iocaton. The shift Is illustrated in the example below.

A B C D E <>
A B C D <> E
A B C <> D E
A B <> C D E

where the three elements <C D E> are shifted to the right. There is an input sequence { <C D E>, <C D>, <C>, <>} and an output sequence {<>, <E>, <D E>, <C D E>}. The "natural" divide and conquer recursive formulation would be to split off one element (say "C" first), trahsfer the rest, and then join the element back. The elements may be produced one at a time, and by producing elements right-to-left they may be consumed immediatedly and require no buffer storage.

We will not derive an in-place element deletion, as it is analogous to the element addition.

7.2 Array structures to represent sets

in the array representation, an ordered set is represented as a contiguous region of an array with increasing array indices giving the storage ordering on the set. There is a one to one, ordered correspondence between elements of the set and the contiguous array elements. This requires one array element or one memory location per set member. Additionally, the boundaries of the set must be marked. One method of boundary marking is to remember the indices of the initial and final elements. The storage of each index is of order log n bits but we will assume one computer "word" per index. We shall use the method of storing the two indices (an example of another boundary marking method is to store special elements just before and after the set), but note that if two regions are adjacent

one boundary marker is sufficient where the boundary index marks the end element of the first set and one before the first element of the second set, so that it may be necessary to add or substract one to find the first element of the second set. (If it were the case that two sets overlapped, i.e. some elements belonged to each set, it would not be possible to merge boundary markers.)

A set represented as a contiguous region of an array may be divided into three parts: the left boundary, the right boundary, and the interior. Each boundary may either be anchored to some array position and thus be immovable or else may move as the set in the sequence of sets grows or shrinks. We shall encounter cases where no boundary is movable, where only one boundary is movable and where both boundaries are movable. Suppose we wish to add an element to a set and re-use all storage locations belonging to the parent set. Then either the right or left boundary must expand outward by one position. If the element is inserted at a movable boundary, then that boundary marker is moved and no other elements in the set need be moved. If the element is inserted in the interior of the set then ail elements to either the right or left of the inserted element must be shifted. If only one boundary marker is movable, then ail elements between the inserted element and the movable boundary marker must be shifted outward. An insertion at a fixed boundary requires the shift of ail elements of the parent set. Thus, if shifting operations are to be minimized the element should be inserted as near to a movable boundary as possible. We may speak of a movable boundary as a growth point and the expansion direction as a growth direction. The case of deletion of an element is analagous to the addition case, i.e. all elements between-the deleted element and a movable boundary must be shifted inward and the boundary reset.

7.3 Location of Sets

Consider where the input and output set sequences will be located and how they will shrink and grow respectively. The analysis of the possibilities is very simple for the sinking and bubble sort (but becomes complicated for quicksort and merge sort). There are only two possibilities for the location of the two sets, input on the left and output on the right or vice-versa. One set cannot grow inside the other since sets must be represented by contiguous regions. Since the left-right distinction does not matter, assume input on the left and output on the right. Each set must have a fixed boundary on the outside and each must share a common movable boundary in the interior. In other words the output sequence will start at the right hand edge and grow toward the left, whereas the input sequence will begin by filling the entire space and shrink toward the left. As each element is transferred the common boundary marker (array index) will move one position to the left.

The situation may be illustrated as follows:

Without further decisions about the nature of the algorithm, we cannot say whether elements will be inserted or deleted at a fixed boundary, interior point, or the movable boundary. However note that insertions or deletions cause no shifting of elements if done at the movable boundary, some shifting if done at the interior of a set, and the most shifting if done at the outside, fixed boundaries.

Accordingly we will strive to minimize shifts by either (a) adding elements at the shared boundary, or-else (b) deleting elements at the shared boundary, when possible. We will see that case (a) is appropriate to a selection sort and case (b) to an insertion sort.

We have now laid the ground work for the in-place insertion sort and the in-place selection sort. in our derivation so far, we have created an in-place transfer sort, where array elements will be shifted appropriately as the elements are transferred from the input set sequence to the output set sequence. It appears that within this set of constraints, there are more than just the selection and insertion possibilities remaining. For example, the largest m elements might be selected out and put in their final position and an insertion sort performed on the next n-m elements. However we have ruled out stranger possibilities such as selecting the first element, finding the final place (by comparing it against all others) inserting it there, finding the final place of the displaced element, etc. This algorithm would allow non-contiguous subset representations.

What constraints lead to a selection or insertion sort? One set of constraints that is adequate is either to always remove elements from the movable boundary or to always place elements at the moveable boundary. This constraint says to allow shifting operations in only the producer or consumer but not both. It also is a uniform algorithm in that it is not a selection for several steps, then an insertion for the rest. Uniformity thus simplifies the resulting algorithm. Another form of constraint is to specify that either (a) the producer re-orders the elements (finds the desired permutation) and the consumer merely stores the elements in that order or (b) the producer merely enumerates the elements and the consumer re-orders or finds the permutation. (One would then set up producer and consumer to minimize shifting,) This form of constraint might arise from some simplicity criterion on the algorithm, i.e. either the producer or the consumer is very simple. The minimal shifting considerations are not applicable to sorting with list structures, but the simplicity consideration is applicable.

7.4 In-place Insertion Sort

Assume that an insertion sort will be synthesized. The producer must generate every element in the input set and transfer each to the output. We may satisfy the constraint of minimizing shifting in the input by removing elements from the boundary position, thus moving the boundary right to left across the input. This enumeration is total since **each** element is visited. This completes the producer.

The consumer or inserter must, according to the inductive hypothesis, keep the output set sorted at each step. Accordingly, the correct position of each new element relative to the other elements in the output must be found. After finding this position, elements to the left of this position are shifted left and the element is inserted. Each set in the output sequence is thus sorted and we have an in-place insertion sort. Now consider the details of finding the correct position and shifting the elements to make room.

The finding of the **correct** position means that the element must be larger than ail to its left and smaller than all **to its** right, According to this definition each position is enumerated, and for each position **the** element is **tested** against **all** others, But since the output set is sorted, **we** know **by transitivity that** the element must be compared only against its two candidate neighbors. Then to enumerate ail positions a linear scan is adequate and the simplest search.

Next a scan direction for the position finding must be selected. We know that a shift is required to make room for the inserted element and that a left to right enumeration will be used for the shift. A heuristic rule for selecting an enumeration direction is to consider the same direction as other scans of the same set. in this case, choosing the same direction as the shifting scan will lead to combining the two scans. Assume we choose **a** left to right

scan for the position finding. The test against its two neighbors can be simplified to a test against only the element to the right since it must be larger then the element to the left or the scan would have ended.

The shifting operation entails shifting left by one ail elements to the left of the inserted element. Recall that the shifting is itself a transfer operation that enumerates each element left-to-right and moves it left by one. Observe that exactly the same set must be enumerated, in exactly the same **order if the transfer is iterative, for both** the **position** finding operation and the insertion operation. **This observation leads to a combining of the** two operations and having only one enumeration; as each element in the output set is produced, it is compared against the inserted element and shifted left if it is smaller. When one is iarger the element is inserted into the vacant position. The resulting algorithm is called an insertion sort.

```
5 3 1 6 2 8 7 4
      .                    severa  steps later

5 3 1 6 2 4 7 8            hold 6 in temporary storage, compare to 2
6           output
5 3 1 2 4 7 8              shift 2, compare 6 to 4
6           outuut
5 3 1 2 4   i s            shift 4, compare 6 to 7, insert 6
            output
5 3 1 2 4 6 7 8
      . output                   finally

1 2 3 4 5 6 7 8
```

SERIES OF SHIFTS AND INSERTS

We observe that a binary chop algorithm for finding the correct position is possible since the output set is ordered. A binary chop would lead to O(n log n) comparisons but still require the same number of shifts, and the algorithm would be more complex,

One final modification brings us to the "sinking" sort: instead of holding the boundary element in any special temporary location, use the newly vacated position to hold it. Thus, at each step, the program performs a comparison to see if the correct position has been found, and if not interchanges, the two elements. (This last modification might not necessarily result in a speed-up, but it clarifies the way in which the insertion class of sorting programs includes the "sinking" sort as a special case.)

5 3 1 6 2 8 7 4	**several steps later**
5 3 1 6 2 4 7 8 output	6 **is to be inserted, compare to** 2, **interchange**
5 3 1 2 6 4 7 8 output	**compare** 6 **to** 4, **interchange**
5 3 1 2 4 6 7 8 output	**compare** 6 **to** 7, **completes insertion of** 6
5 3 1 2 4 6 7 8 output	**finally**
1 2 3 4 5 6 7 8	

SINKING--SERIES Of INTERCHANGES

7.5 **In-place** Selection Sort

Assume. that a selection sort will be synthesized. To minimize shifting operations, elements must be inserted into the output set at the movable boundary, so that they fill up the output set linearly, right to left. Also, the inductive hypothesis requires that the output set be sorted at each step as it is built up. Together, **these** force the elements to be produced, largest element first, then the rest in descending order. Thus the selection process is a series of operations, each having three steps: find the largest element, delete it from the input, then insert it into the output at the boundary between the sets.

First we will synthesize an algorithm to find the largest element. Our derivation will be very simple compared to an implemented version. The simplest "find" algorithm for an enumerable set is to generate each element and then test to see whether it is the largest element. This requires $O(m^2)$ comparisons per selection, where m is the number of elements in the current set of the input sequence, since each element must be compared against all others.

It is possible to speed up this algorithm to m comparisons. In the basic form of generate and test, each element is compared against all others and if it is not larger, the next element is attempted as a candidate. By the simple rule of stopping an enumeration once the answer is found, we need not compare the element with any more once a larger element is found. This is still an order $O(m^2)$ algorithm, but a little better, The refinement to O(m) follows.

Assume that a fixed, left-to-right linear scan is chosen as the generation order of candidates. This is the obvious choice since it is the least complex total enumeration. Next, assume that the same fixed left-to-right enumeration order is chosen for enumerating the elements to which the candidate is compared. Any other fixed order would do as long as both are the same. Perhaps the strongest a priori reason for attempting to make both enumeration orders the same is the resulting uniformity, which sometimes results in a simpler algorithm. The heuristic of choosing the same enumeration order will also lead to combining two loops.

Next consider the step at which the candidate "a" is compared to an element, say "c", and found to be smaller. By the rule of stopping an enumeration as soon as possible, no other elements are compared to "a". According to our algorithm we select as next candidate the element "b" just to the right of "a".

< . . . a b . . . c >

Here we invoke the rule of starting an enumeration as late as possible. By transitivity since **"c"** is larger than **"a"** and **"a"** is larger than **"b",** then **"c"** is larger than **"b".** Hence **"b"** need not be considered, This argument applies to all elements between **"a"** and **"c",** so that none be considered as candidates. However **"c"** is not known to be smaller than any element to its right, so **"c"** becomes the next candidate. This completes the derivation of an algorithm for finding the largest element in m steps. Note that we have effectively derived a special case of ttie more general rule **"to** find the extremum, perform one total enumeration and replace the candidate element each time a better element is found." The more general rule could have been used in this case, but we anticipate that the ability to derive it should also prove useful in other program synthesis situations.

The next operation is that of deleting the largest element from the input set. One deletion method is to remove the largest element and shift left all elements to its right. As before in the insertion sort, the shifting and the comparison operations are both linear, left to right. The only difference is that the element being rippled along changes when a new candidate for the largest element is found. This series of exchanges re-orders the input set, but there is no constraint that the input order not be changed. Thus, as in the insertion sort, the shifting can be interleaved with the search for the largest element, to form a series of interchanges to both find the largest element and delete it by moving it to the right. The final position of the largest element is at the left side of the output set, thus also completing the insertion into the output set. This algorithm is called bubble sort or exchange selection. In this case where it is permissable to re-order the input set, another simple deletion and insertion method can be used. The largest element is just interchanged with the boundary element of the input set. This algorithm is called straight selection sort. We note that the

bubble sort does more work than just selecting the largest element in each scan since more than one element may be moved toward its destination. The selection sort can be made O(n log n) comparisons by using a tree-selection of the largest element as in heapsort.

8. Equal-size Split

And now we return to the partitioning sorts where the sets are divided into subsets that may each have more than one element. Suppose that we choose to divide the initial set into two subsets. The methods discussed for two subsets are probably extendable to several subsets. The next question is, "by what critierion shall we split the set into two parts"? We shall consider criteria which divide the set into two approximately equal sets. This often leads to faster algorithms, in particular the equal size split sometimes speeds up an algorithm from $O(n^2)$ to $O(n \log n)$ comparisons. The depth of recursion is reduced from n to log n and if the combined split and join operations are held to O(n) comparisons, then this speed up is achieved.

One way is to split the set into a left-part and right-part depending upon the position, but not the value, of the elements. Or it can be split into all elements larger than some size and all smaller than some size, which depends upon the value of the elements rather than the position. This choice is the major factor in determining what type of sort routine is produced.

If the set is divided into two sets each respectively having all elements larger than and smaller than a given size, the result is a class including quicksort. This class may be thought of as a general form of selection sort where the set having ail elements larger (say) than a given size is being selected, i.e. a set rather than an element is being selected. Most of the burden. fails on the split operation, to select out these sets. The join operation is a simple append (things may be viewed slightly differently for the conventional in-place quicksorts).

If the set is divided arbitrarily, into two parts (say a left and right part for convenience) the split operation is simple, and the work of ordering the elements is done in

the join operation. This method leads to mergesort and may be thought of as a general form of insertion sort (although, to complicate matters, a subpart of one form of the **merge** sort can be viewed as a selection sort).

8.1 Quicksort

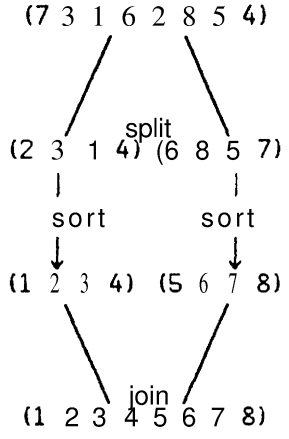
Let us assume that we shall divide the set into all elements larger than some number and all less than or equal to that number. Let these two subsets be called LARGE and SMALL. Next, we must decide on the size of the dividing number. Our preferred choice is a number that divides the set so that LARGE and SMALL are of approximately equal size. if such a size is known then we may use it. if not, then we must make some estimate of this median, A random number might be a first guess. Better, we can select randomly but limited to any number within the range of the set being sorted (although not necessarily in the set, since it may fall between two numbers in the set). This at least increases the likelihood of it dividing the set equally. An easy way to find a number within the range spanned by the set is to choose an element of the set. Let us make this choice, but note that more effort could produce a better estimation of the mid-point by sampling some elements of the set.

We will call the chosen element DIVIDER. Which element in the set shall be the DIVIDER? if there is no reason to believe the set is ordered in any way then pick the most convenient element. (If the set is perhaps somewhat ordered, then we might select from the middle of the set.) A convenient element to choose, in most data structures is often the first or last element. Let us choose the first element, and let the rest of the set be called REST.

I-low shall this element be used to divide the set? The simplest method is to compare it against each of the other elements in the **set.** Those that are larger go into LARGE and those

that are less than or equal go into SMALL. The splitting process requires a total enumeration over the finite set REST, or in other words a total generate and process algorithm where the process is the addition of the element to one of two subsets. Note that, as in the bubble and sinking sorts, the sets SMALL, LARGE, and REST are really sequences of sets, each growing or shrinking one element at a time. At this level of abstraction, we have completed the split operation. After completing the entire sort at this level, we shall continue in more detail. Most of the detail is concerned with efficiency and particular data structures.

Recall that the three top-level steps were to split, sort the subsets, and then join, Assume that the sorting of the subsets will be done recursively by the same algorithm. Now for the join. Since both subsets are sorted, and all the elements in LARGE are larger than those in SMALL, we may join the subsets with an append. So the join is quite simple. (In fact, if an in-place sort is used, then the two sets are already in their correct relative positions and no explicit join operation is required.) This completes the high-level description, as illustrated below:



8.1 .1 Details of the Split Operation

This is the part of quicksort which is most interesting. We shall show a sequence of methods of doing the dividing, each of which typically results in greater efficiency. Each element of REST must be compared to DIVIDER, and put into its proper subset, What order of enumeration shall be used? Left-to-right or right-to-left linear scans are the easiest for most data structures.

As before we shall attempt to re-use space to produce an in place sort. When the algorithm was described abstractly, no mention was made of whether the two sets LARGE and SMALL fit into the original space or were placed in new spaces. If *every* set in the sequences LARGE and SMALL is put into a new space, then the average space requirement is about 2n log n. But we will be able to achieve an in-place sort requiring roughly n locations. The argument that this is possible is the same as that given for singleton split and insertion sets. Examination of the algorithm shows that each set in the SMALL, LARGE, and REST sequences need be remembered only until the next set is produced. Thus after an element is produced from REST it can be placed in SMALL or LARGE and deleted from REST. The total number of spaces needed at any time is just n, plus some bookkeeping overhead.

As before we shall assume that the sets will be placed in contiguous regions of arrays. Since each subset will be in a contiguous region of an array, each region can be marked by a left and right boundary rather than by marking each element. Also, two boundaries are adequate to mark the remainder of the original set. Where two regions touch, the two boundaries merge into one boundary and less marking space will be required. Some additional space will be needed to hold the element being transferred and to save the state of enumeration of elements of REST.

Now consider the process of enumerating the set and placing each element into its subset. There are several things needed; an enumeration order, e.g. left to right, or right to left, or something more complex, and a place for each subset. We will need to choose an initial position for each set, and growth directions for each boundary of each set. There are six possible relative positions for the three sets, LARGE, SMALL, and REST. Since an array can be accessed equally well from either direction, a left-right reversal of position is effectively the same structure. So, by symmetry, we need only consider. 3 possibilities. If we also take advantage of the symmetry of SMALL and LARGE, two possibilities are left:

Case 1: SMALL LARGE REST
Case 2: SMALL REST LARGE

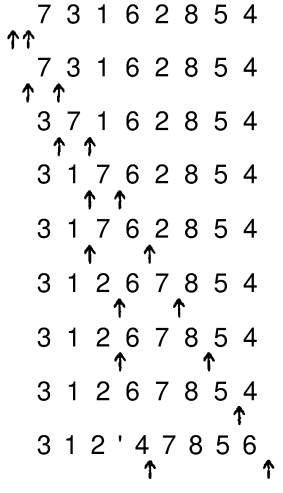
with all other cases essentially similar.

Recall that each element will be compared with the DIVIDER element and assigned to its subset. No delay is required to make the decision and place the element since the choice of a location to insert it' is not dependent upon any element not yet transferred. Thus the element may be consumed as soon as it is produced. As soon as the produced element is removed, then a vacant space exists and there are enough places so that there is room, but the problem is to have that room occur at the right place to yield an efficient algorithm. There are no constraints on the order of elements in LARGE and SMALL so the element may be inserted any place. But in order to minimize shifting operations it should be inserted at a movable boundary. Also, insertion at a boundary minimizes search for an insertion position. But in order to minimize shifts by inserting at the boundary position there must be space made available at the boundary. What does it cost to make space available at the boundaries for each of the two cases?

Case 1: SMALL LARGE REST

There is no constraint on the enumeration order from REST, but if left-to-right is chosen then elements that need to be inserted into LARGE can be added at the LARGE-REST boundary and no shifting is required.

Now consider, however, insertion of elements into SMALL. Any interior position or the left boundary position of SMALL necessitates shifts within SMALL and requires one new space on its right, so the best insertion position is at its right boundary. Since no element is given up from that boundary position, LARGE must be shifted right by one element, The obvious way is to shift all elements right by one, but this is expensive. By noting that there is no ordering constraint on LARGE, it can be shifted right by moving only its leftmost element to its rightmost boundary. This algorithm is illustrated below. This is not a bad algorithm, but case 2 yields an interesting and possibly more efficient algorithm.



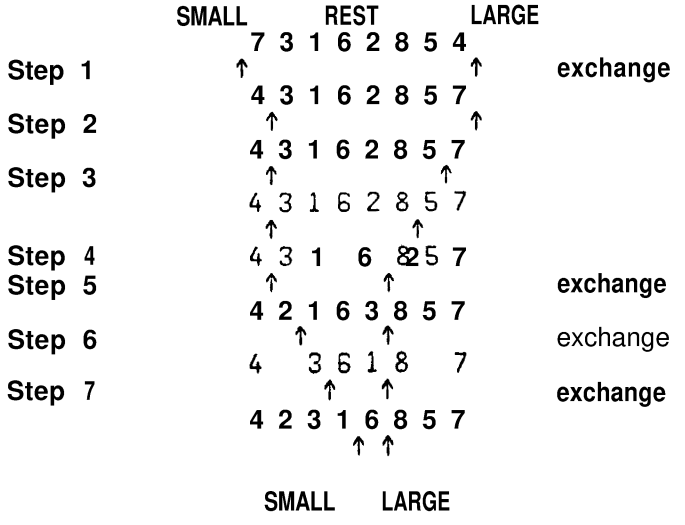
SMALL LARGE REST

This completes the synthesis for case 1. Now we proceed to case 2.

Case 2: **SMALL REST LARGE**

Both LARGE and SMALL share **a** boundary with REST. Any complete enumeration of the elements in REST **will** work. Merely take each element from rest and place it at the growth boundary of LARGE or SMALL respectively. Place the removed boundary element of REST **into the vacated** position. This strategy will in general re-order the set REST, which is allowable. In the cases where **a** boundary element in REST is next to **its** proper place in LARGE or SMALL no interchange takes place and only the boundary marker is moved.

Let us show one example



The moving boundaries are marked by arrows. Our enumeration strategy was to enumerate from right to left, with SMALL to the left and LARGE to **the** right. The element **"4"** was the divider, At each step the element being compared to 4 is just to the left of the right most arrow. Note that after an interchange, the same *position* in REST had to be re-examined since it held a new element. When all of REST is enumerated, we are done. One way to check is to note that both SMALL and LARGE boundaries meet.

Next let us consider a change in the quicksort algorithm. Note that an element could sometimes be moved to a temporary location, then later examined and moved again. To see this, notice that in step 5 of the example, 2 and 3 were interchanged, although the element 3 was really in a suitable position. We could have just changed the boundary of small to include it, rather than moving it twice. Can this unncessary move be avoided in general? So far we see no necessity for it in terms of the amount of space available to consume a produced element.

A solution is to consider an alternative enumeration order for the set REST that allows the proper interchange spaces to become available as needed. The proper spaces occur at both boundaries of REST. So a good strategy is to enumerate both left-to-right and right-to-left in some reasonable manner. Suppose we move right-to-left. Sometimes just the boundary will change (when the element belongs to LARGE) and sometimes an element will have to be moved (when the element belongs to SMALL). When an element is to be moved, begin an enumeration from left-to-right, but don't make the exchange yet. Moving from left to right, again, sometimes an element will stay, as it is placed in SMALL by moving the boundary. But sometimes we will encounter an element that needs to be moved into LARGE. Then when both need to be moved, make an interchange and begin again. Whether we begin left-to-right or vice-versa doesn't matter as long as we make no interchanges until both elements need to be interchanged. We illustrate by example.

```
                REST
          7 3 1 6 2 8 5 4          begin
SMALL-, ↑               ↑ ←LARGE
          7 3 1 6 2 8 5 4          scan right to left, note 4 should be moved
        ↑       ↑
          7 3 1 6 2 8 5 4          scan left to right, note 7 should be moved
            ↑       ↑
          4 3 1 6 2 8 5 7          interchange 4 and 7
            ↑       ↑
          4 3 1 6 2 8 5 7          scan right to left
            ↑     ↑
          4 3 1 6 2 8 5 7          scan right to left
            ↑   ↑
          4 3 1 6 2 8 5 7          scan right to left, note 2 should be moved
            ↑ ↑
          4 3 1 6 2 8 5 7          scan left to right
              ↑ ↑
          4 3 1 6 2 8 5 7          scan left to right
                ↑↑
          4 3 1 6 2 8 5 7          scan left to right, note 6 should be moved
                ↑↑
          4 3 1 2 6 8 5 7          interchange 2 and 6
```

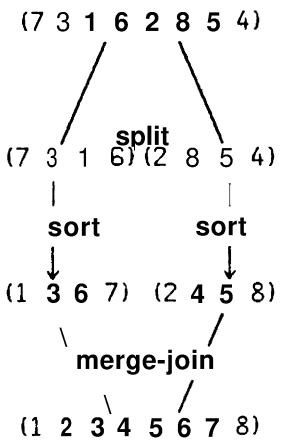
So only two interchanges and no shifts were necessary with the new scanning strategy.

8.2 Mergesort

Recall that the work could be done in the split or in the join, In this section we shall assume that the work is done in the join and a simple split operation will be used. It does not matter which elements go into which of the two subsets. So the splitting method should be the easiest possible. If an array is used as the data structure, then a division into a right and left half requires storage space for only one boundary marker, an index. To find the value of this index only a division is required. If linked-lists are used, the division into two sets requires one scan and possibly another scan to find the length.

We wiil assume that an array is divided into right and left halves. Each of these

subsets, say S_1 and S_2, is then sorted. Now, how are the two to be joined? A simple append will not work since elements in either one may be larger than elements in the other one, Thus some merging technique will be required as illustrated **below:**



in the quicksort case the union of the two sorted subsets reduced to a simple concatenation operation. For mergesort, we must find another technique. **A** form of the divide-and-conquer paradigm may be employed. Assume the two sorted subsets are S_1 and S_2. We wish to join them into S_3 by an ordered set union operation, i.e. $S_3 = S_1 \cup S_2$. By the divide-and-conquer paradigm, we may split $S_1 \cup S_2$ into two parts, form the union of the remainder, and then joih them. Obviously a split into S_1 and S_2 leaves us with the same problem and we are no closer to a solution. So consider a singleton split of one element and the rest, say into {a} and $(S_1 - \{a\}) \cup S_2$. After forming the union of $S_1 - \{a\}$ and $S_2, \{a\}$ is joined.

. Note that {a} can be selected as the extremum, in which case the split is doing the work or else the element {a} is chosen for the simplicity of the split and the join does the work.

One case appears to be a selection-style ordered union and one appears to be an insertion style ordered union. Also, both may be converted to an iterative transfer version,

rather than a recursive version. We shall not discuss that conversion but we will trace the synthesis for both the insertion and the selection case, assuming an iterative transfer.

8.2.1 insertion paradigm for merge

Consider the insertion process. Each element from $S_1 \cup S_2$ must be inserted into S_3. The order of selection of elements from S_1 and S_2 is not constrained. By observing that S_2 is sorted, just, let $S_2 = S_3$ effectively transferring all elements from S_2 in S_3 at no cost. Then successive elements of S_1 must be inserted into S_2. The process of finding an efficient join or merge operation may be seen as **a** sequence of speed-ups of the various enumerations during the production of an element from S_1 and the generation of its correct position in S_2. As in our insertion sorts discussed earlier, use can be made of the knowledge that the *element-consuming* set S_2 remains sorted during the insertion process. in addition we can use the information that the *element-producing* set is sorted to find still further speed-ups.

First consider the insertion of an element into S_2. Since S_2 is sorted, in general not every position. need be examined. In fact, for a linear scan of positions, as soon as the first position is encountered in which the element to be inserted is larger than the element to its left and smaller than the element to its right, then no further searching is necessary, since that position is correct and the element may be inserted there, The correctness of this position follows from the transitivity of the sorted set. The test for correctness may also be slightly optimized, **as** discussed earlier.

Next consider the enumeration order of elements from S_1. if they are generated linearly, from, say left-to-right, then each successive element is larger than the last. This

means that in scanning S_2, the scan need not begin to the left of the recently inserted element, since by transitivity it must be larger than ail of those. So the scan need only begin at the position to the right of the inserted element. This method requires a state-saving scheme for the consuming set S_2 to remember the previous insertion position. This process turns out to be linear in the number of comparisons of elements. By observing that the depth of recursion is log n, the total number of comparisons is of order n log n.

The merge sort can be done in place. The determination of the feasibility of an in place sort is the same as shown for the other sorts discussed earlier. However, shifting will be required in S_2 since the elements are inserted into the interior of S_2. Suppose the sets are placed so that S_1 is on the left and S_2 is on the right. Then all elements in S_2 to the left of the inserted element must be shifted since S_2 cannot be re-ordered. The rule that says to minimize shifting by taking elements from the common boundary may be invoked to suggest a right-to-left descending order of elements in S_1. Since elements are taken from the boundary, no shifts in S_1 are required. However, there may be as many as $n/2$ shifts of $n/2$ elements required for S_2. The insertion paradigm fits well with list representation of sets.

8.2.2 Selection paradigm for merge

We now present an alternate derivation path, a *selection* paradigm. in the selection paradigm the enumeration order is forced to select elements from both S_1 and S_2 according to the final ordering. Thus first the largest is selected, then the next largest, and so on. These are then inserted in a linear order one after the other.

The two sorted subsets, S_1 and S_2, are the input to the selection process and the

output, say S_3, will be the sorted set. If no a priori knowledge about the ordering relation on S, and S_2 is used, then all elements are enumerated to see which is the largest. As discussed earlier an inefficient way is to compare each against all others. A reasonably simple speedup is to carry a best-so-far candidate along, and replace it with any larger element. By transitivity, this finds the largest element in a number of comparisons equal to the size of the input, say n. Thus, a scan of the entire set produces one element. To produce all elements of S_3 requires about $n^2/2$ comparisons.

The fact that we know both S_1 and S_2 are sorted leads to one more speedup. The largest element in S_1 is its last element. The largest element in S_2 is its last element. The largest element in $S_1 \cup S_2$ must be either the largest in S_1 or the largest in S_2. If this is not clear, suppose that "e", the last element of S_1 is larger than the last element of S_2. We know that "e" is larger than all of S_1. Now, by transitivity, since it is larger than the last element of S_2, it is larger than all elements of S_2. So to find the largest element, only the two last elements need be compared. Now produce that element, remove it from its parent set, say S_1, and put it in the output set S_3. Again the same situation holds; the largest element is now either the new last element of S_1 or the last element of S_2. Compare these two elements to produce the second element and remove it to the output set. We see that this results in one comparison per output element, so that the number of comparisons is reduced to O(n), from $O(n^2)$. The speedup resulted from the use of the knowledge that the input sets were already sorted.

Note that two enumerations are being carried out simultaneously on S_1 and S_2, and the sequencing between the two enumerations depends on the data. We used removal of each element as the state-saving scheme, but other schemes such as marking elements or moving list pointers would also be adequate. The selection paradigm makes clear that if a second

set of n locations is available, say S_3, then a sort with no shifting is possible since S_3 is created in order.

9. Conclusions

In this paper we have attempted to explicate the programming knowledge for space re-utilization, ordered set enumeration, and the divide-and-conquer paradigm. With this knowledge converted into rule form one could synthesize efficient sort programs, ordered set unions and other programs.

The principal derivation path--divide-and-conquer, then singleton or equal size split, then recursive to iterative transfer, then in-place, then shifting or exchange--proved to be satisfactory. In particular we found this approach simpler than considering exchange sorts as a separate class. The paradigm could probably be reasonably extended to include heapsort, radix sort and others, with the introduction of trees and other primitives. However, while we felt the paradigm was adecluate, we explored few other canadidates except for transfer and exchange paradigms.

Much of the knowledge expressed here and in our previous papers has been implemented as rules and has been tested in programs that successfully synthesized many sorting and other programs. However, we have not yet implemented many of the higher-level rules and assume that further embellishments will be required.

An interesting and more global research question is that of the utility of this particular approach to program synthesis. At one extreme of the synthesis spectrum is the macro expansion of templates, or the instantiation of parameters into existing pieces of' code. Those techniques are rigid but computationally cheap and are adequate for many purposes such as compiler code generators for medium-level languages. At the other extreme are more computationally expensive theorem proving techniques that search larger spaces of

possible programs. Such techniques are more likely to create "new" algorithms or at least unanticipated ones, but at the possibly prohibitive cost of considering many unusable programs. Our approach is more of a middle ground but as suits a particular application it may be combined with other techniques.

10. Acknowledgements

We would like to acknowledge helpful discussions and suggestions from Elaine Kant, Bill **Laaser, Maarten van Emden,** John Darlington, Bob Floyd, Don Knuth, and Leo Guibas.

11. References

[1] Barstow, David R. *A knowledge-based system for automatic program construction.* Proceedings of the Fifth International Joint Conference on Artificial intelligence, Cambridge, Massachusetts, August 1977, pages 382-388.

[2] Barstow, David R., and Kant, Elaine. *Observations on the interaction of coding and efficiency knowledge in the PSI program synthesis system.* Proceedings of the Second International Conference on Software Engineering, San Francisco, California, October 1976, pages 19-31.

[3] Burstall, R. M., and Darlington, John. *A transformation system for developing recursive programs,* Journal of the ACM, 24, 1977, pages 44-67.

[4] Darlington, John. *A synthesis of several sorting algorithms,* Research Report 23, Department of Artificial Intelligence, University of Edinburgh, Scotland, July 1976.

[5] Ginsparg, Jerrold. *A parser for English and its application in an automatic programming system.* Ph.D. thesis, Stanford University, 1977.

[7] Green, C. Cordell. *A summary of the PSI program synthesis system.* Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August 1977, pages 380-381.

[6] Green, C. Cordell. *The design of the PSI program synthesis system.* Proceedings of the Second international Conference on Software Engineering, San Francisco, California, October 1976, pages 4-l 8.

[8] Green, C. Cordell, and Barstow, David R. *A hypothetical dialogue exhibiting* a *knowledge base for* a *program understanding system.* in Elcock, E. W., and Michie, D. (Eds.). Machine Representations of Knowledge. Ellis Horwood Ltd. and John Wylie, 1976.

[9] Green, C. Cordell, and Barstow, David R. *Some rules for the automatic synthesis of programs,* Advance Papers of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi, Georgia, USSR, September 1975, pages 232-239.

[10] Kant, Elaine. *The selection of efficient implementations for* a *high level language,* Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices, Volume 12, Number 8, SIGART Newsletter, Number 64, August 1977, pages 140-l 46.

[11] Knuth, Donald E. *Sorting and searching.* The Art of Computer Programming, Volume 3, Addison-Wesley, Reading Massachusetts.

[12] Laker, William. *Synthesis of recursive programs.* Personal communication.

[13] Ludlow, Juan. Masters Project. Stanford University, 1977.

[14] Manna, Zohar and Waldinger, Richard. *The automatic synthesis of recursive programs.* Proceedings of A C M SIGART-SIGPLAN Symposium on **Artificial Intelligence and Programming Languages,** SIGPLAN Notices, Volume 12, Number 8, SIGART Newsletter, Number 64, August 1977, pages 29-33.

[15] McCune, Brian P. *The PSI program model builder: synthesis of very high-level programs,* Proceedings of ACM SIGART-SIGPLAN **Symposium on Artificial Intelligence and Programming Languages,** SIGPLAN Notices, Volume 12, Number 8, SIGART Newsletter, Number 64, August 1977, pages 130-l 39.

[17] Phillips, Jorge. *A framework for the synthesis of programs from traces using multiple knowledge sources.* Proceedings of the Fifth international Joint **Conference on Artificial Intelligence,** Cambridge, Massachusetts, August 1977, page 812.

[18] Sedgewick. *Quicksort,* STAN-CS-75-492, Stanford, California, May 1975.

12. Appendix

In this appendix we illustrate how a few of the rules may be stated in more detail. A typical and widely-used rule is one which enumerates all the elements in an explicitly-stored set, and performs some operation upon each element. An English level description of the rule is:

In order to write an enumerator for an explicitly stored set,
(a) determine the order for generating the elements,
(b) select an appropriate scheme for saving the state of the enumeration between the production of the elements,
(c) based upon the enumeration order and the state-saving scheme, write the bocly, initializer, incrementer and termination test.

This rule might be invoked by a higher level rule with certain constraints such as (a) "the enumeration is to be total", i.e. all elements must be processed or (b) a data structure may already have been selected or (c) an enumeration order may already have been imposed. Suppose that no enumeration order has been specified, but the enumeration is constrained to be total and the set is stored in an array. Then one of the rules for enumeration orders must be chosen. We will assume that the following rule is selected:

```
(PROP←  ENUMERATION-ORDER
        (ENUMERATE-POSITIONS)
        (QUOTE STORED) )
```

which may be paraphrased as "An enumeration order for an enumeration of positions in a sequential collection is the stored order."

After the enumeration order has been chosen, a state-saving scheme must be selected. This involves the invocation of three rules. The first is:

```
(REF←   (ENUMERATION-STATE
            (#P ENUMERAT I ON
                (#P ENUMERATION-ORDER (?#= STORED)))
            (#P COLLECTION (←← X)))
    .(#NEW  POSITION-IN-COLLECTION
            (←#P COLLECTION X)))
```

which may be paraphrased as "The enumeration state of an enumeration whose enumeration order is the stored order may be represented as a position in the collection." (REF← denotes the refinement of one concept into another. P refers to various attributes of such concepts.) The second is:

```
. (REF←   (POSITION-IN-COLLECTION
            (#P COLLECTION (←← X)
                (#REF ARRAY)))
          (#NEW ITEM-INDEX
            (←#P ARRAY X)))
```

which says "If a collection is represented as an array, a position in that collection may be represented as an index in that array." The final rule based on the assumption that the output code-is to be in the LISP language:

```
(REF←   (ITEM-INDEX)
        (#NEW LISP-INTEGER) )
```

which states "An index in an array may be represented as a LISP integer."

Other state saving schemes can be pointers, bit strings, properly list marks, hash table entries, list removal, element overwriting, etc. Some are suitable for non-destructive enumerations, others only for destructive enumerations. Some are adequate for non-linear enumeration orders, others are not.

Next it is necessary to write the body, the initializer, and the termination test. Assume that the collection under consideration is represented as an array. Then the termination test synthesis process invokes a series of rules as follows:

```
(REF←    (TEST-ENUMERATION-TERMINATION
             (#P STATE (←← S))
             (#P COLLECTION (←← C)))
         (#NEW TEST-FINAL-ENUMERATION-STATE
             (←#P STATE S)
             (←#P COLLECTION C)))
```

which may. be read as "A test of enumeration termination may be refined into a test on whether the state-saving scheme is in its final state."

```
(REF←    (TEST-FINAL-ENUYERATION-STATE
             (#P STATE (←← S)
                 (#P ENUMERATION
                     (#P RANGE (?#= TOTAL))))
             (#P COLLECTION (←← C)))
         (#NEW  TEST-ALL-ELEMENTS-ENUMERATED
             (←#P STATE S)
             (←#P COLLECTION C)))
```

Or, "If an enumeration is total, a test of whether a state-saving scheme is in its final state may be refined into a test of whether the state indicates that all elements have been enumerated." (Notice how the above rule checks to see whether the range of the enumeration is constrained, as we assumed earlier. Were there no such constraint, then other rules to determine an appropriate range would have to be considered.)

```
(REF←    (TEST-ALL-ELEMENTS-ENUMERATED
             (#P STATE (←← S)
                 (#P ENUMERATION
                     (#P ENUMERATION-ORDER (?#= STORED)))
                 (#RDS(#REF POSITION-IN-COLLECTION111
             (#P COLLECTION (←← C)))
         (#NEW TEST-POSITION-AFTER-LAST-ITEM
             (←#P POSITION P)
             (←#P COLLECTION C)))
```

"If the enumeration order is the stored order and the state is saved as a position in the collection, a test of whether the state indicates that all elements have been enumerated may be refined into a test of whether the position indicates the position after the last item in the collection."

Notice that up to this point, all of the rules have been general enough to deal with linked lists as well as arrays. The next rule is specific to arrays:

```
(REF←    (TEST-POSITION-AFTER-LAST-ITEM
             (#P COLLECTION (←← C)
                 (#RDS (#REF ARRAY) ))
             (#P POSITION (←← P)
                 (#RDS (#REF ITEM-INDEX))))
         (#NEW TEST-INDEX-AFTER-LAST-I TEM
             (←#P ARRAY C)
             (←#P INDEX P)))
```

which may be paraphrased, "if the collection is represented as an array and the position is represented as an index into the array, then a test of whether the position indicates the position after the last item may be refined into a test of whether the index is the index after the last item."

```
(REF←    (TEST-INDEX-AFTER-LAST-ITEM
             (#P ARRAY (←← A))
             (#P INDEX (←← L)))
         (#NEW GREATER-THAN
             (←#P ARG1 L)
             (←#P ARG2
                 (#NEW GET-UPPER-BOUND
                     (←#P ARRAY A)))))
```

which may be read, "A test of whether an index is the index after the last item may be refined into a test of whether the index is greater than the upper bound of the **array."**

The final rule in the sequence produces a call to a LISP function:

```
(REF←    (GREATER-THAN
             (#P ARG1 (←← A1)
                 (#RDS (#REF LISP-INTEGER)))
             (#P ARG2 (←← A2)
                 (#RDS (#REF LISP-INTEGER))))
         (#NEW LISP-FUNCTION-CALL
             (←#P FUNCTION-NAME (QUOTE IGREATERP))
             (←#P ARGUMENTS (LIST A1 A2))))
```

"A test of whether a LISP integer is greater than another LISP integer may be refined into a call to the LISP function **IGREATERP."**

Notice that it is still necessary to write the code which will retrieve the upper bound of the array. In most cases this would result in the retrieval of the value of a bound variable, so the final LISP expression **would** be: "(IGREATERP STATE UPPERBOUND)", where STATE is the variable which holds the state and UPPERBOUND is the variable which holds the upper bound of the array. Despite the detail, the above derivation is itself a slight over-simplification, in that an array is simply a correspondence between integers and values, a fact which would be included in a complete derivation.

Notice also, that for an list enumeration with arbitrary ordering, and using deletion as a state saving scheme, a very different piece of code would be produced. The synthesis of the other parts of the enumerator follow similar lines. The body becomes a refinement of whatever-operation is being applied to each element. The **incrementer** can be complex, as in the enumerator for the selector of a selection sort of very simple as in the enumerator of the the selector for an insertion sort.

After the pieces of code for the various parts of the enumerator have been produced it is relatively sraightforward to assemble them into a program and clean up the resulting code. Most of our programs have been produced in LISP but we now have a small number of rules for producing SAIL code added by Juan Ludlow-Saldivar [13].