An Experiment in Knowledge-based Automatic Programming

David R. Barstow*

Yale University, New Haven, CT 06520, U.S.A.

ABSTRACT

Human programmers seem to know a lot about programming. This suggests a way to try to build automatic programming systems: encode this knowledge in some machine-usable form. In order to test the viability of this approach, knowledge about elementary symbolic programming has been codified into a set of about four hundred detailed rules, and a system, called PECOS, has been built for applying these rules to the task of implementing abstract algorithms. The implementation techniques covered by the rules include the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings, as well as several techniques for enumerating the elements of a collection. The generality of the rules is suggested by the variety of domains in which PECOS has successfully implemented abstract algorithms, including simple symbolic programming, sorting, graph theory, and even simple number theory. In each case, PECOS's knowledge of different techniques enabled the construction of several alternative implementations. In addition, the rules can be used to explain such programming tricks as the use of property list markings to perform an intersection of two linked lists in linear time. Extrapolating from PECOS's knowledge-based approach and from three other approaches to automatic programming (deductive, transformational, high level language), the future of automatic programming seems to involve a changing role for deduction and a range of positions on the generality-power spectrum.

1. Introduction

1.1 Motivation

The experiment discussed here is based on a simple observation: human programmers seem to know a lot about programming. While it is difficult to state precisely what this knowledge is, several characteristics can be identified. First, human programmers know about a wide variety of concepts. Some of these concepts are rather abstract (e.g., set, node in a graph, sorting, enumeration, pattern matching), while others are relatively concrete (e.g., linked list, integer, conditional, while loop). Second, much of this knowledge deals with specific implementation techniques (e.g., property list markings to represent a set, bucket hashing to represent a mapping, quicksort, binary search). Third, programmers know guidelines or heuristics suggesting when these implementation techniques may be appropriate (e.g., property list markings are inappropriate for sets if the elements are to be enumerated frequently). In addition to these kinds of rather specific knowledge, programmers also seem to know several general strategies or principles (e.g., divide and conquer), which can be applied in a variety of situations. Finally, although programmers often know several different programming languages, much of their knowledge seems to be independent of any particular language.

Is this knowledge precise enough to be used effectively by a machine in performing programming tasks? If not, can it be made precise enough? If so, what might such an automatic programming system be like? The experiment discussed here was designed to shed some light on questions like these. The experimental technique was to select a particular programming domain, elementary symbolic programming, and a particular programming task, the implementation of abstract algorithms, and to try to codify the knowledge needed for the domain and task. For reasons to be discussed later, the form used to express the knowledge was a set of rules, each intended to embody one small fact about elementary symbolic programming. A computer system, called PECOS, was then built for applying such rules to the task of implementing abstract algorithms.

The resulting knowledge base consists of about 400 rules dealing with a variety of symbolic programming concepts. The most abstract concepts are collections[1] and mappings, along with the appropriate operations (e.g., testing for membership in a collection, computing the inverse image of an object under a mapping) and control structures (e.g., enumerating the objects in a collection). The implementation techniques covered by the rules include the representation of collections as linked lists, arrays (both ordered and unordered), and Boolean mappings, the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings (indexed by range element). PECOS writes programs in LISP (specifically, INTERLISP [29]); while some of the rules are specific to LISP, most (about three-fourths) are independent of LISP or any other target language. In addition to the rules concerned with details of the different implementation techniques, PECOS has about a dozen choice-making heuristics dealing with the appropriateness and relative efficiency of the techniques. None of PECOS's rules are concerned with general strategies such as divide and conquer.

The utility of the rules is suggested by the variety of domains in which PECOS

* This research was conducted while the author was affiliated with the Computer Science Department, Stanford University, Stanford, California, 94305. The research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-76-C-0206. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U.S. Government.

[1] The term "collection" is used since the rules do not distinguish between multisets, which may have repeated elements, and sets, which may not.

was able to implement abstract algorithms, including elementary symbolic programming (simple classification and concept formation algorithms), sorting (several versions of selection and insertion sort), graph theory (a reachability algorithm), and even simple number theory (a prime number algorithm). PECOS's knowledge about different implementation techniques enabled the construction of a variety of alternative implementations of each algorithm, often with significantly different efficiency characteristics.

PECOS has also been used as the Coding Expert of the PSI (Ψ) program synthesis system [13]. Through interaction with the user, Ψ's acquisition phase produces a high level description (in PECOS's specification language) of the desired program. The synthesis phase, consisting of the Coding Expert (PECOS) and the Efficiency Expert (LIBRA [17]), produces an efficient LISP implementation of the user's program. In this process, PECOS can be seen as a "plausible move generator", for which LIBRA acts as an "evaluation function". The nature of the search space produced by PECOS, as well as techniques for choosing a path in that space, will be discussed further in Section 6.1.

1.2. Representations of programming knowledge

Unfortunately, most currently available sources of programming knowledge (e.g., books and articles) lack the precision required for effective use by a machine. The descriptions are often informal, with details omitted and assumptions unstated. Human readers can generally deal with the informality, filling in the details when necessary, and (usually) sharing the same background of assumptions. Before this programming knowledge can be made available to machines, it must be made more precise: the assumptions must be made explicit and the details must be filled in.

Several different machine-usable forms for this knowledge are plausible. Some kind of parameterized templates for standard algorithms is one possibility, and would work well in certain situations, but would probably not be very useful when a problem does not fit precisely the class of problems for which the template was designed. In order to apply the knowledge in different situations, a machine needs some "understanding" of why and how the basic algorithm works. Alternatively, one could imagine some embodiment of general programming principles, which could then be applied in a wider variety of situations. However, such a technique loses much of the power that human programmers gain from their detailed knowledge about dealing with particular situations. The form used in this experiment is something of a middle ground between these two extremes: the knowledge is encoded as a large set of relatively small facts. Each is intended to embody a single specific detail about elementary symbolic programming. Ultimately, of course, automatic programming systems will need knowledge from many places on the power-generality spectrum.

There are still several possible forms for these facts, ranging from explicit formal axioms about the relevant concepts and relations, to explicit but less formal symbolic rules such as those of MYCIN [26], to the implicit form of code embedded within a program designed to perform the task. For this experiment, the form of symbolic rules was selected.[2] Several rules from PECOS's knowledge base are given below (for clarity, English paraphrases of the internal representation are used; the internal representation will be discussed in Section 5.2):

> *A collection may be represented as a mapping of objects to Boolean values; the default range object is* FALSE.
>
> *If the enumeration order is linear with respect to the stored order, the state of an enumeration may be represented as a location in the sequential collection.*
>
> *If a collection is input, its representation may be converted into any other representation before further processing.*
>
> *If a linked list is represented as a* LISP *list without a special header cell, then a retrieval of the first element in the list may be implemented as a call to the function* CAR.
>
> *An association table whose keys are integers from a fixed range may be represented as an array subregion.*

The primary reason for using the symbolic rule representation, as opposed to a mathematical axiomatization of the relevant concepts and relations, is simply that the relevant concepts and relations are not well enough understood (in many cases, not even identified) for an axiomatization to be possible. Ultimately, it may be possible to axiomatize the concepts and relations, but a necessary first step is to identify and understand them. A second reason is that most of the human-oriented sources of programming knowledge are not particularly mathematical in nature, and these sources are the first places to try when looking for programming knowledge. Finally, the development of other rule-based systems has provided considerable knowledge engineering experience that greatly facilitated PECOS's development.

1.3. The value of an explicit rule set

A large part of this experiment involved developing a set of rules about symbolic programming. The rule set itself provides several benefits. First, precision has been added to the human-oriented forms of programming knowledge, in terms of both the unstated assumptions that have been made explicit and the details that have been filled in. For example, the rule given above about representing a collection as a Boolean mapping is a fact that most programmers know; it concerns the characteristic function of a set. Without knowing this rule, or something similar,

[2] Actually, "fact" may be a better term than "rule", but "rule" will be used throughout because of the similarity between PECOS and other "rule-based" systems.

it is almost impossible to understand why a bitstring (or even property list markings) can be used to represent a set. Yet this rule is generally left unstated in discussions of bitstring representations; the author and the reader share this background knowledge, so it need not be stated. As another example, consider the rule given above about representing an association table as an array subregion. The fact that an array is simply a way to represent a mapping of integers to arbitrary values is well known and usually stated explicitly. The detail that the integers must be from a fixed range is usually not stated. Yet if the integers do not satisfy this constraint, an array is the wrong representation, and something like a hash table should be used.

A second major value of PECOS's rule set is the identification of useful programming concepts. Consider, for example, the concept of a sequential collection: a linearly ordered group of locations in which the elements of a collection can be stored. Since there is no constraint on how the linear ordering is implemented, the concept can be seen as an abstraction of both linked lists and arrays. At the same time, the concept is more concrete than that of a collection. One benefit of such intermediate-level concepts is a certain economy of knowledge: much of what programmers know about linked lists and arrays is common to both, and hence can be represented as rules about sequential collections, rather than as one rule set for linked lists and one for arrays. For example, the following two pieces of LISP code are quite similar:

```
(PROG (L)
      (SETQ L C1)
  RPT (COND ((NULL L)
             (RETURN NIL)))
      (RPLACA L (IPLUS (CAR L) 1))
      (SETQ L (CDR L))
      (GO RPT))

(PROG (I)
      (SETQ I 1)
  RPT (COND ((IGREATERP I (ARRAYSIZE C2))
             (RETURN NIL)))
      (SETA C2 I (IPLUS (ELT C2 I) 1))
      (SETQ I (IPLUS I 1))
      (GO RPT))
```

Each adds 1 to every element of a collection. In the first, the collection C1 is represented as a linked list; in the second, the collection C2 is represented as an array. The code is similar because both involve enumerating the objects in a sequential collection. The state of the enumeration is saved as a location in the collection (here, either a pointer L or an integer I). The remaining aspects depend on the representation of the enumeration state: an initial location (here, either C1, a pointer, or 1, an index), a termination test (either (NULL I) or (IGREATERP I (ARRAYSIZE C2))), an incrementation to the next location (either (CDR L) or (IPLUS I 1)), and a way to find the object in a given location (either the CAR of L or the array entry for I in C2). Other than the differences based on the representation of locations, all of the knowledge needed to write these two pieces of code is independent of the way the sequential collection is represented.

This example also illustrates the identification of particular design decisions involved in programming. One of the decisions involved in building an enumerator of the objects in a sequential collection is selecting the order in which they should be enumerated. In both of the above cases, the enumeration order is the "natural" order from first to last in the sequential collection. This decision is often made only implicitly. For example, the use of the LISP function MAPC to enumerate the objects in a list implicitly assumes that the stored order is the right order in which to enumerate them. While this is often correct, there are times when some other order is desired. For example, the selector of a selection sort involves enumerating the objects according to a particular ordering relation.

A final benefit of PECOS's rules is that they provide a certain kind of explanatory power. Consider, for example, the well-known (but little documented) trick for computing the intersection of two linked lists in linear time: map down the first list and put a special mark on the property list of each element; then map down the second collecting only those elements whose property lists contain the special mark. This technique can be explained using the following four of PECOS's rules (in addition to the rules about representing collections as linked lists):

> *The intersection of two collections may be implemented by enumerating the objects in one and collecting those which are members of the other.*
>
> *If a collection is input, its representation may be converted into any other representation before further processing.*
>
> *A collection may be represented as a mapping of objects to Boolean values; the default range object is* FALSE.
>
> *A mapping whose domain objects are atoms may be represented using property list markings.*

Given these rules, it can be seen that the trick works by first converting the representation of one collection from a linked list to property list markings with Boolean values, and then computing the intersection in the standard way, except that a membership test for property list markings involves a call to GETPROP rather than a scan down a linked list.[3]

[3] Since a new property name is created (via GENSYM) each time the conversion routine is executed, there is no need to erase marks after the intersection is computed, except to retrieve the otherwise wasted space.

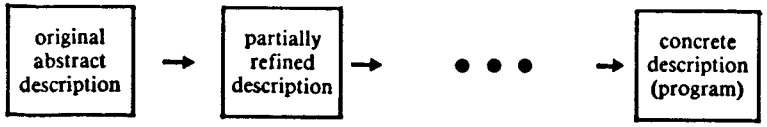
As another example, consider the use of association lists: lists whose elements are dotted pairs (generally, each CAR in a given list is unique). In some situations such a structure should be viewed simply as a collection represented as a list; in others it should be viewed as a way to represent a mapping. The following rule clarifies the relationship between these two views:

> *A mapping may be represented as a collection whose elements are pairs with a "domain object" part and a "range object" part.*

Thus, in general an association list should be viewed as a mapping, but when implementing particular operations on the mapping, one must implement certain collection operations. For example, retrieving the image of a given domain object involves enumerating the elements of the list, testing the CAR of each pair for equality with the desired domain object. In turn, implementing this search involves rules about sequential collections, enumeration orders, and state-saving schemes such as those mentioned above. Thus, PECOS's rules are sufficient for writing the definition of the LISP function ASSOC.

1.4. Program construction through gradual refinement

PECOS constructs programs through a process of gradual refinement. This process may be simply illustrated as a sequence of program descriptions:



Each description in the sequence is slightly more refined (concrete) than the previous description. The first is the program description in the specification language and the last is the fully implemented program in the target language. Each refinement step is made by applying one of the rules from PECOS's knowledge base, thereby transforming the description slightly. When several rules are relevant in the same situation, PECOS can apply each rule separately. In this way PECOS can construct several different implementations from one specification. This capability for developing different implementations in parallel is used extensively in the interaction between PECOS and LIBRA in Ψ's synthesis phase.

2. Overview of the Knowledge Base

A detailed discussion of PECOS's entire rule set is beyond the scope of this paper and the interested reader is referred elsewhere [3]. Nevertheless, a brief overview may help to clarify what PECOS can and cannot do.

2.1. General rules and LISP rules

The rules can be divided into "general" and "LISP-specific" categories, where the latter deal with such concepts as CONS cells and function calls. Of PECOS's four hundred rules, slightly over one hundred are LISP-specific.[4] Most of the LISP rules are quite straightforward, merely stating that specific actions can be performed by specific LISP functions. Note that knowledge about LISP is associated with the uses to which the LISP constructs can be put. That is, rather than describing the function CAR in terms of axioms or pre- and post-conditions, as is done in most automatic programming and problem solving systems, PECOS has rules dealing with specific uses of CAR, such as returning the item stored in a cell of a "LISP list" or returning the object stored in one of the fields of a record structure represented as a CONS cell. Thus, there is never a necessity of searching through the knowledge base of facts about LISP in order to see whether some function will achieve some desired result; that information is stored with the description of the result. This representation reduces searching significantly, but also lessens the possibilities of "inventing" some new use for a particular LISP function.

The rest of this overview will be concerned only with the "general" rules. Three major categories of implementation techniques are covered by the rules: representation techniques for collections, enumeration techniques for collections, and representation techniques for mappings. The rules also deal with several lower-level aspects of symbolic programming, but they will be omitted completely from this discussion.

2.2. Representation of collections

Conceptually, a collection is a structure consisting of any number of substructures, each an instance of the same generic description. (As noted earlier, PECOS's rules do not distinguish between sets and multisets.) The diagram in Fig. 1 summarizes the representation techniques that PECOS currently employs for collections, as well as several (indicated by dashed lines) that it does not. Each branch in the diagram represents a refinement relationship. For example, a sequential collection may be refined into either a linked list or an array subregion. These refinement relationships are stored in the knowledge base as refinement rules. Of course, the diagram doesn't indicate all of the details that are included in the rules (e.g., that an array subregion includes lower and upper bounds as well as some allocated space).

As can be seen in the diagram of Fig. 1, PECOS knows primarily about the use of Boolean mappings and sequential collections. Both of these general techniques will be illustrated in the derivation of the Reachability Program in Section 4. Although "distributed collection" occurs in a dashed box, PECOS can implement a collection using property list markings by following a path through a Boolean mapping to a distributed mapping. The most significant missing representations

[4] In a preliminary experiment, the LISP-specific rules were replaced by rules for SAIL (an ALGOL-like language [24]), and PECOS wrote a few small SAIL programs [22]. The programs were too simple to justify definite conclusions, but they are an encouraging sign that this distinction between "general" and "language-specific" rules is valid and useful.

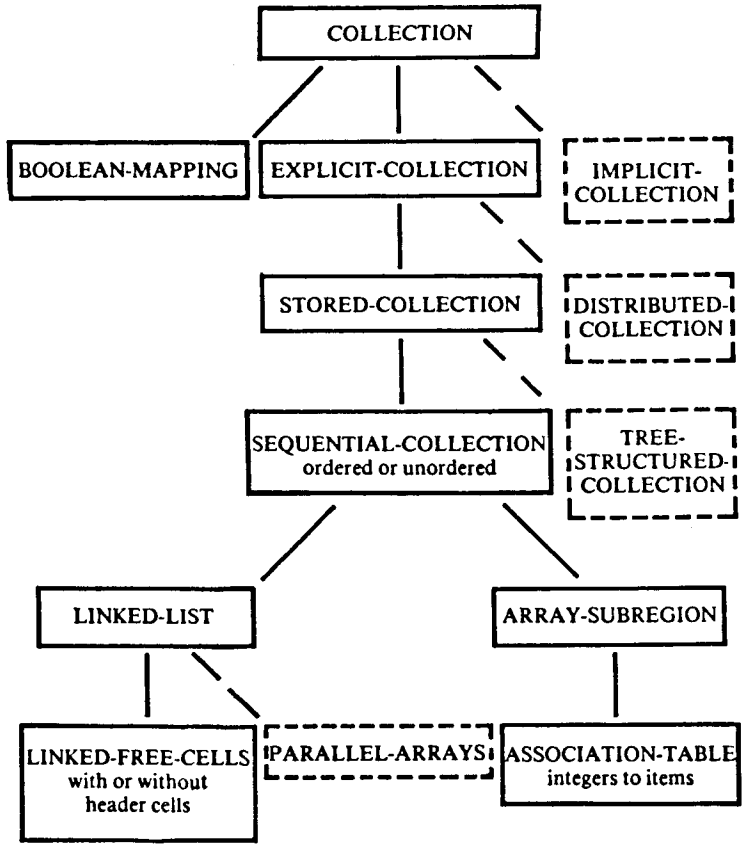


FIG. 1.

are the use of trees (such as AVL trees or 2–3 trees) and implicit collections (such as lower and upper bounds to represent a collection of integers). Codification of knowledge about these techniques would be a valuable extension of PECOS's knowledge base.

Note the extensive use of intermediate-level abstractions. For example, there are four concepts between "collection" and "linked free cells". As noted earlier, such intermediate levels help to economize on the amount of knowledge that must be represented, and also facilitate choice making.

2.3. Enumerations over stored collections

In its most general form, enumerating the elements of a collection may be viewed as an independent process or coroutine. The elements are produced one after another, one element per call. The process must guarantee that every element will be produced on some call and that each will be produced only once. In addition, there must be some way to indicate that all of the elements have been produced, as well as some way to start up the process initially. The process of constructing an enumerator for a stored collection involves two principal decisions: selecting an appropriate order for enumerating the elements, and selecting a way to save the state of the enumeration.

There are several possible orders in which the elements can be produced. If the enumeration order is constrained to be according to some ordering relation, then clearly that order should be selected. If it is unconstrained, a reasonable choice is to use the stored (first-to-last) order, either from the first cell to the last (for linked lists) or in order of increasing index (for arrays). In some cases, it may be useful to use the opposite (last-to-first) order.

The enumeration state provides a way for the enumerator to remember which elements have been produced and which have not. There are many ways to save such a state. Whenever the enumeration order is first-to-last (or last-to-first), an indicator of the current position is adequate: all elements before (or after, for last-to-first) the current position have been produced and all elements after (before) the position have not. PECOS's rules handle these cases, as well as the case in which the enumeration order is constrained and the collection is kept ordered according to the same constraint, in which case a position indicator is also adequate for saving the state.

The situation is somewhat more complex for nonlinear enumerations (i.e., the enumeration order is not the same as the stored order or its opposite); finding the next element typically involves some kind of search or scan of the entire collection. During such a search, the state must be interrogated somehow to determine whether the element under consideration has already been produced. There are basically two kinds of nonlinear enumeration states, destructive and nondestructive. PECOS's rules deal with one destructive technique, the removal of the element from the collection. A technique not covered by the rules is to overwrite the element. The rules also do not cover any nondestructive techniques.

2.4. Representation of mappings

A mapping is a way of associating objects in one set (range elements) with objects in another set (domain elements).[5] A mapping may (optionally) have a default image: if there is no stored image for a particular domain element, a request to determine its image can return the default image. For example, when a Boolean mapping is used to represent a collection, the default image is FALSE.

The diagram of Fig. 2 summarizes representation techniques for mappings. As with collection representations, there are several intermediate levels of abstraction for mappings. Note that an association list representation is determined by following the path from "mapping" to a "collection" whose elements are domain/range pairs; the refinement path in the collection diagram given earlier then leads to a

[5] PECOS's rules only deal with many-to-one mappings and not with more general correspondences or relations.

linked list of pairs. Property lists give a distributed mapping whose domain is the set of atoms. A plex (or record structure) with several fields would constitute a mapping whose domain is the set of field names. The most significant mapping representations missing from PECOS's rules are implicit mappings (such as function definitions) and discrimination nets. As with the use of trees to represent collections, codifying knowledge about discrimination nets will certainly involve several aspects of graph algorithms. The rules currently deal with only one small aspect of hash tables: the use of INTERLISP's hash arrays. This is clearly another area where further codification would be valuable.

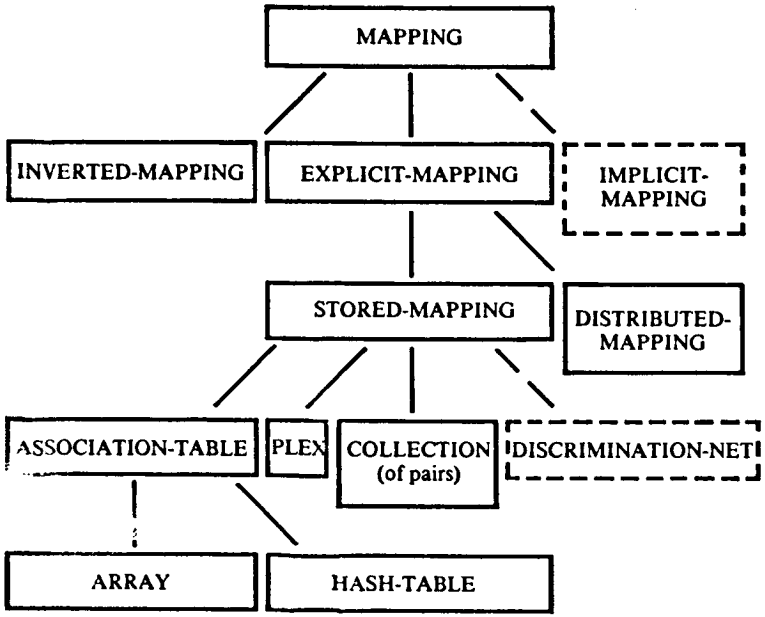


FIG. 2.

3. Sample Programs

As an indication of the rang. of topics covered by PECOS's rules, five sample programs will be presented in this section.[6] The next section gives a detailed look at a sixth. The first four of these were selected as target programs early in the research, in order to have a focus for the development of the rules. After most of the rules were written, the last two were selected as a way of testing the generality of the rules. About a dozen rules (dealing primarily with numeric operations) needed to be added for the last two programs to be constructed.

[6] Theoretically PECOS can implement any algorithm that can be described in its specification language. In practice, however, PECOS cannot handle specifications much longer than "a page" before space limitations become prohibitive.

3.1. Membership test

The variety of implementations that PECOS can produce is illustrated well by a simple membership test. PECOS can implement such a test in about a dozen ways, differing primarily in the way that the collection is represented. If a sequential collection is used, there are several possibilities. In the special case of a linked list, the LISP function MEMBER can be used. In addition, there are various ways of searching that are applicable for either linked lists or arrays. If the collection is ordered, the search can be terminated early when an element larger than the desired element is found. If the collection is unordered, the enumeration must run to completion. A rather strange case is an ordered enumeration of an unordered collection, which gives a membership test whose time requirement is $O(n^2)$. If the collection is represented as a Boolean mapping, a membership test is implemented as the retrieval of the image of the desired element. For each way to represent a mapping, there is a way to retrieve this image. The LISP functions GETHASH, GETPROP, and ELT apply to hash arrays, property list markings, and arrays respectively. In addition, a collection of pairs can be searched for the entry whose CAR is the desired element and the entry's CDR can be returned. PECOS has successfully implemented all of these cases.

3.2. A simple concept classification program

The second target program was a simple classification program called CLASS. CLASS inputs a set (called the concept) and then repeatedly inputs other sets (called scenes) and classifies them on the basis of whether or not the scene fits the concept. A scene fits a concept if every member of the concept is a member of the scene. The specification given to PECOS is paraphrased below[7]:

Data structures

CONCEPT	a collection of integers
SCENE	a collection of integers or "QUIT"

Algorithm

```
CONCEPT ← input a list of integers;
loop:
    SCENE ← input a list of integers or the string "QUIT";
    if SCENE = "QUIT" then exit the loop;
    if CONCEPT is a subset of SCENE
        then output the message "Fit"
        else output the message "Didn't fit";
    repeat;
```

[7] Integers were used as the elements of the scenes and concept to facilitate the use of ordered collections. A different set of implementations would be possible with different types of elements in the sets.

The major variations in implementations of CLASS involve different representations for SCENE and the role they play in the subset test. The test is refined into an enumeration of the elements of CONCEPT, searching for one that is not a member of SCENE.[8] In the simplest case, the internal representation of SCENE is the same as the input representation, a linked list. The other cases involve converting SCENE into some other representation before performing the subset test. The major motivation for such a conversion is that membership tests for other representations are much faster than for unordered linked lists. One possibility is to sort the list, but the time savings in the membership test may not be sufficient to offset the time required to perform the sorting.[9] Other possibilities include the use of Boolean mappings such as property list markings and hash tables. PECOS has successfully constructed all of these variations.

3.3. A simple formation program

The third target program was TF, a rather simplified version of Winston's concept formation program [32]. TF builds up an internal model of a concept by repeatedly reading in scenes which may or may not be instances of the concept. For each scene, TF determines whether the scene fits the internal model of the concept and verifies the result with the user. The internal model is then updated based on whether or not the result was correct. The internal model consists of a set of relations, each marked as being necessary or possible. A scene fits the model if all of the necessary relations are in the scene. The updating process is divided into four cases: (1) if the model fit the scene and this was correct (indicated by user feedback), all relations in the scene that are not in the model are added to the model and labelled "possible"; (2) if the model fit but this was incorrect, any relation marked "possible" but not in the scene is picked and relabelled as "necessary"; (3) if the model did not fit and this was incorrect, all relations marked "necessary" that are not in the scene are relabelled as "possible"; (4) otherwise, there is no change.

The most interesting variations in the implementation revolve around the representation of the mapping CONCEPT. Inverting this mapping gives two sets to be represented, NECESSARY and POSSIBLE. Since "any" and "all" operations are applied to these sets, a stored collection is appropriate (although for some distributions of input data Boolean mapping representations may be better). Since elements will be added and removed from both sets, linked lists are reasonable representations. The computation of the domain of CONCEPT is fairly interesting as the domain set does not exist explicitly with inverted mappings, but must be computed (in this case by a union of NECESSARY and POSSIBLE).

[8] For some representations that PECOS cannot handle, other forms for the subset test are appropriate. For example, if CONCEPT and SCENE are both represented as bit vectors, "CONCEPT $\wedge \neg$ SCENE" is zero if and only if CONCEPT is a subset of SCENE.

[9] PECOS cannot currently use the technique of sorting both lists so that they can be scanned in parallel, thereby greatly increasing the savings.

Note, however, that the only operation applied to the domain is a membership test. In such a case, the test can be refined into a disjunction of two membership tests, one on NECESSARY and one on POSSIBLE, and there is no need to explicitly compute the domain of CONCEPT. This is the implementation that PECOS constructed.

3.4. Sorting

PECOS's development originally began as an investigation into the programming knowledge involved in simple sorting programs [14–16]. PECOS's current rule set is sufficient to synthesize a variety of sorting algorithms within the transfer paradigm, in which sorting is viewed as a process of transferring the elements from the (unordered) input collection one at a time to the (ordered) output collection. Under this view, the part of the program that selects the next element to transfer is simply an enumerator over the elements of the input set. If the enumeration order of this selector is the same as the stored order of the input, the resulting sort program does an insertion sort. If the enumeration order of the selector is the same as (or the opposite of) the sorted order, the program is a selection sort. PECOS has implemented selection and insertion sort programs using both arrays and lists for the input and output collections. Thus, PECOS can carry out (approximately) the reasoning required for what was earlier described as a "hypothetical dialogue" [15].

3.5. Primes

The following problem is taken from Knuth's textbook series [18]:

> 7.1–32. [22] (R. Gale and V. R. Pratt.) The following algorithm can be used to determine all odd prime numbers less than N, making use of sets S and C.
>
> **P1.** [Initialize] Set $j \leftarrow 3$, $C \leftarrow S \leftarrow \{1\}$. (Variable j will run through the odd numbers 3, 5, 7, ... At step P2 we will have
>
> $$C = \{n | n \text{ odd}, 1 \leqslant n < N, n \text{ not prime, and } \mathrm{gpf}(n) \leqslant p(j)\},$$
> $$S = \{n | n \text{ odd}, 1 \leqslant n < N/p(j), \text{ and } \mathrm{gpf}(n) \leqslant p(j)\},$$
>
> where $p(j)$ is the largest prime less than j and $\mathrm{gpf}(n)$ is the greatest prime factor of n; $\mathrm{gpf}(1) = 1$.)
>
> **P2.** [Done?] If $j \geqslant N/3$, the algorithm terminates (and C contains all the nonprime odd numbers less than N).
>
> **P3.** [Nonprime?] If $j \in C$ then go to step P5.
>
> **P4.** [Update the sets.] For all elements n in S do the following: If $nj < N$ then insert nj into S and into C, otherwise delete n from S. (Repeat this process until all elements n of S have been handled, including those which were just newly inserted.) Then delete j from C.

P5. [Advance j.] Increase j by 2 and return to P2.

Show how to represent the sets in this algorithm so that the total running time to determine all primes $< N$ is $O(N)$. Rewrite the above algorithm at a lower level (i.e., not referring to sets) using your representation.

[Notes: The number of set operations performed in the algorithm is easily seen to be $O(N)$, since each odd number $n < N$ is inserted into S at most once, namely when $j = \text{gpf}(n)$, and deleted from S at most once. Furthermore we are implicitly assuming that the multiplication of n times j in step P4 takes O(1) units of time. Therefore you must simply show how to represent the sets so that each operation needed by the algorithm takes O(1) steps on a random-access computer.]

Since PECOS's rules do not cover enumerations over collections that are being modified during the enumeration, a slightly modified version of the original algorithm was given to PECOS. In this version, the set S has been replaced by two sets $S1$ and $S2$, and the set P is created to output the set of primes (the complement of C).

Data structures

C	a set of integers
P	a set of integers
$S1$	a set of integers
$S2$	a set of integers
J	an integer
K	an integer
N	an integer

Algorithm

```
N← input an integer;
J← 3;
C← {1};
S1← {1};
loop:
  if 3*J > N then exit;
  if J is not a member of C then
    S2← S1;
    S1← { };
    loop until S2 is empty:
      for any X in S2:
        remove X from S2;
        if J*X < N then
          add X to S1;
          add J*X to S2;
          add J*X to C;
        remove J from C;
    J← J+2;
    repeat;
  K← 3;
  P← { };
  loop:
    if N < K then exit;
    if K is not a member of C
      then add K to P;
    K← K+2;
    repeat;
  output P as a linked list.
```

The only operations being performed on $S2$ are addition, removal, and taking "any" element. The "any" operation suggests that a Boolean mapping may be inappropriate and the frequent destructive operations suggest that an array may be relatively expensive. Thus, an unordered linked list is a reasonable selection. Since the value of $S1$ is assigned to $S2$, a representation conversion can be avoided by using the same representation for both sets. This is especially useful here, since the only operation applied to $S1$, the addition of elements, is relatively simple with unordered linked lists. The only operations applied to C are addition, removal, and two membership tests. Such operations are fairly fast with Boolean mappings. Since the domain elements of the mapping are integers with a relatively high density in their range of possible values, an array of Boolean values is a reasonable representation of C. PECOS has implemented the Primes Program in this way, as well as with a linked list representation for C. To check the relative efficiency of the two implementations, each was timed for various values of N, see Table 1. (Note the approximately linear behaviour of the Boolean array case and the distinctly nonlinear behavior of the linked list case.)

TABLE 1. (Times are given in milliseconds)

N	C as linked list	C as Boolean array
10	0.05	0.04
50	0.28	0.20
100	0.63	0.40
500	6.40	2.02
1000	21.21	4.08

4. A Detailed Example

Perhaps the best way to understand how PECOS works is to see an example of the use of programming rules and the refinement paradigm to construct a particular program. This example also demonstrates that the rules enable PECOS to deal

with the program at a very detailed level and that the same rules may be used in several different situations.

In order to focus on the nature of the rules and the refinement process, the example will be presented in English. After a description of the abstract algorithm to be implemented, several specific aspects of it will be discussed in detail. For each of these aspects, the abstract description of that part of the algorithm will be presented, followed by a sequence of rules, together with the refinements they produce in the original description. The result of this sequence of rule applications will be a LISP implementation of the original abstract description.

4.1. The reachability problem

The example is based on a variant of the reachability problem [30]:

Given a directed graph, G, and an initial vertex, v, find the vertices reachable from v by following zero or more arcs.

The problem can be solved with the following algorithm:

Mark v as a boundary vertex and mark the rest of the vertices of G as unexplored. If there are any vertices marked as boundary vertices, select one, mark it as explored, and mark each of its unexplored successors as a boundary vertex. Repeat until there are no more boundary vertices. The set of vertices marked as explored is the desired set of reachable vertices.

Note that the algorithm's major actions involve manipulating a mapping of vertices to markings.

Based on this observation, the algorithm can be expressed at the level of PECOS's specification language. The following is an English paraphrase of the specification given to PECOS when this example was run. (As a notational convenience, $X[Y]$ will be used to denote the image of Y under the mapping X and $X^{-1}[Z]$ will be used to denote the inverse image of Z under X.)

Data structures

VERTICES	a collection of integers
SUCCESSORS	a mapping of integers to collections of integers
START	an integer
MARKS	a mapping of integers to {"EXPLORED", "BOUNDARY", "UNEXPLORED"}

Algorithm

```
VERTICES ← input a list of integers;
SUCCESSORS ← input an association list of ⟨integer, list of integers⟩ pairs;
START ← input an integer;
for all X in VERTICES:
  MARKS[X] ← "UNEXPLORED";
MARKS[START] ← "BOUNDARY";
repeat until MARKS⁻¹["BOUNDARY"] is empty:
  X ← any element of MARKS⁻¹["BOUNDARY"];
  MARKS[X] ← "EXPLORED";
  for all Y in SUCCESSORS[X]:
    if MARKS[Y] = "UNEXPLORED" then MARKS[Y] ← "BOUNDARY";
output MARKS⁻¹["EXPLORED"] as a list of integers.
```


The specification is abstract enough that several significantly different implementations are possible. For example, MARKS could be represented as an association list of ⟨integer, mark⟩ pairs or as an array whose entries are the marks. The relative efficiency of these implementations varies considerably with several factors. For example, if the set of vertices (integers) is relatively sparse in a large range of possible values, then implementing MARKS as an array with a separate index for each possible value would probably require too much space, and an association list would be preferable. On the other hand, if the set of vertices is dense or the range small, an array might allow much faster algorithms because of the random-access capabilities of arrays. For the remainder of this discussion, it will be assumed that the range of possible values for the vertices is small enough that array representations are feasible. Note also that concrete input representations are specified for VERTICES (a linked list), SUCCESSORS (an association list), and START (an integer), and that an output representation is specified for MARKS^{-1}["EXPLORED"] (a linked list). These constrain the input and output but not the internal representation. They are intended to reflect the desires of some hypothetical user and PECOS could handle other input and output representations equally well.

When PECOS was run on the Reachability Algorithm, there were several dozen situations in which more than one rule was applicable. In most of these cases, selecting different rules would result in the construction of different implementations, and PECOS has successfully implemented the algorithm in several different ways. In the following discussion, one particular implementation is synthesized. About two-thirds of the choices made during the synthesis were handled by PECOS's choice-making heuristics, and in the remaining third, a rule was selected interactively in order to construct this particular implementation.

4.2. SUCCESSORS

Under the SUCCESSORS mapping, the image of a vertex is the set of immediate successors of the vertex:

$$\text{SUCCESSORS}[v] = \{x \mid v \rightarrow x \text{ in } G\}$$

SUCCESSORS is constrained to be an association list when it is input, but such a representation may require significant amounts of searching to compute SUCCESSORS[X]. Since this would be done in the inner loop, a significantly faster algorithm can be achieved by using an array representation with the entry at index k being the set of successors of vertex k. In the rest of this section, the derivation of this array representation will be considered in detail.

4.2.1. Representation of SUCCESSORS

SUCCESSORS is a mapping of integers to collections of integers. An English paraphrase of PECOS's internal representation is given below:

SUCCESSORS:
MAPPING (integers → collections of integers)

The selection of an array representation for SUCCESSORS involves four distinct decisions: that each association in the mapping be represented explicitly, that the associations be stored in a single structure, that a tabular structure be used, and that an array be used for the table. These four decisions are made by applying a sequence of four rules (corresponding to the path from "mapping" to "array" in the diagram of mapping representations given earlier). Each rule results in a slight refinement of the abstract description of SUCCESSORS:

A mapping may be represented explicitly.

SUCCESSORS:
EXPLICIT MAPPING (integers → collections of integers)

An explicit mapping may be stored in a single structure.

SUCCESSORS:
STORED MAPPING (integers → collections of integers)

A stored mapping with typical domain element X and typical range element Y may be represented with an association table whose typical key is X and whose typical value is Y.

$SUCCESSORS_{table}$:
ASSOCIATION TABLE (integers → collections of integers)

(Subscripts, as in $SUCCESSORS_{table}$, are used to distinguish between representations at different refinement levels.)

An association table whose typical key is an integer from a fixed range and whose typical value is Y may be represented as an array with typical entry Y.

$SUCCESSORS_{array}$:
ARRAY (collection of integers)

The final step involves selecting a particular data structure in the target language, in this case INTERLISP's array representation:

An array may be represented directly as a LISP *array.*

$SUCCESSORS_{lisp}$:
LISP ARRAY (collection of integers)

The objects stored in the array must also be represented. Through a sequence of six rule applications, tracing the path in the collection diagram from "collection" to "linked free cells", followed by a LISP-specific rule, a LISP list representation is developed:

$SUCCESSORS_{lisp}$:
LISP ARRAY (*LISP LIST* (integer))

4.2.2. *SUCCESSORS*[*X*]

Determining the set of successor vertices for a given vertex involves computing the image of that vertex under the SUCCESSORS mapping. The abstract specification of this operation is:

compute the image of *X* under SUCCESSORS

The construction of the program for computing SUCCESSORS[*X*] follows a line parallel to the determination of the representation of SUCCESSORS:

If a mapping is stored as an association table, the image of a domain element X may be computed by retrieving the table entry associated with the key X.

retrieve the entry in $SUCCESSORS_{table}$ for the key *X*

If an association table is represented by an array, the entry for a key X may be retrieved by retrieving the array entry whose index is X.

retrieve the entry in $SUCCESSORS_{array}$ for the index *X*

If an array is represented as a LISP *array, the entry for an index X may be retrieved by applying the function ELT.*

(ELT $SUCCESSORS_{lisp}$ *X*)

4.2.3. *Converting between Representations of SUCCESSORS*

Recall that the input representation for SUCCESSORS is constrained to be an association list of ⟨integer, list of integers⟩ pairs:

$SUCCESSORS_{input}$:
LISP LIST (*CONS CELL* (DOMAIN . RANGE))
DOMAIN: integer
RANGE: *LISP LIST* (integer)

Since the input and internal representations differ, a representation conversion must be performed when the association list is input. The description for the input operation is as follows:

```
SUCCESSORS ← input a mapping (as an association list);
```

The following rule introduces the representation conversion:

If a mapping is input, its representation may be converted into any other representation before further processing.

```
SUCCESSORSinput ← input a mapping (as an association list);
SUCCESSORS ← convert SUCCESSORSinput
```

The conversion operation depends on the input representation:

If a mapping is represented as a stored collection of pairs, it may be converted by considering all pairs in the collection and setting the image (under the new mapping) of the "domain object" part of the pair to be the "range object" part.

```
for all X in SUCCESSORSinput:
    set SUCCESSORS[X:DOMAIN] to X:RANGE
```

(Here X:DOMAIN and X:RANGE signify the retrieval of the "domain object" and "range object" parts of the pair X.) Since the pairs in $SUCCESSORS_{input}$ are represented as CONS cells, the X:DOMAIN and X:RANGE operations may be implemented easily through the application of one rule in each case.

If a pair is represented as a CONS cell and part X is stored in the CAR part of the cell, the value of part X may be retrieved by applying the function CAR.
If a pair is represented as a CONS cell and part X is stored in the CDR part of the cell, the value of part X may be retrieved by applying the function CDR.

```
for all X in SUCCESSORSinput:
  set SUCCESSORS[(CAR X)] to (CDR X)
```

The implementation of the "set SUCCESSORS[(CAR X)]" operation is constructed by applying a sequence of rules similar to those used for implementing SUCCESSORS[X] in the previous section, resulting in the following LISP code:

```
for all X in SUCCESSORSinput:
  (SETA SUCCESSORSlisp (CAR X) (CDR X))
```

In constructing the program for the "for all" construct, the first decision is to perform the action one element at a time, rather than in parallel:

An operation of performing some action for all elements of a stored collection may be implemented by a total enumeration of the elements, applying the action to each element as it is enumerated.

```
enumerate X in SUCCESSORSinput:
  (SETA SUCCESSORSlisp (CAR X) (CDR X))
```

Constructing an enumeration involves selecting an enumeration order and a state-saving scheme:

If the enumeration order is unconstrained, the elements of a sequential collection may be enumerated in the order in which they are stored.
If a sequential collection is represented as a linked list and the enumeration order is the stored order, the state of the enumeration may be saved as a pointer to the list cell of the next element.

The derivation now proceeds through several steps based on this particular state-saving scheme, including the determination of the initial state (a pointer to the first cell), a termination test (the LISP function NULL), and an incrementation step (the LISP function CDR):

```
STATE ← SUCCESSORSinput;
loop:
  if (NULL STATE) then exit;
  X ← (CAR STATE);
  (SETA SUCCESSORSlisp (CAR X) (CDR X));
  STATE ← (CDR STATE);
  repeat;
```

The complete LISP code for this part is given below, exactly as produced by PECOS. The variables V0074, V0077, V0071, and V0070 correspond to $SUCCESSORS_{input}$, STATE, X, and $SUCCESSORS_{lisp}$, respectively.

```
(PROG (V0077 V0075 V0074 V0071 V0070)
      (PROGN (PROGN (SETQ V0074 (PROGN (PRIN1 "Links:")
                                       (READ)))
                    (SETQ V0070 (ARRAY 100)))
             (SETQ V0077 V0074))
G0079
      [PROGN (SETQ V0075 V0077)
             (COND
               ((NULL V0077) (GO L0078)))
             (PROGN (PROGN (SETQ V0071 (CAR V0075))
                           (SETA V0070 (CAR V0071)
                                       (CDR V0071)))
                    (SETQ V0077 (CDR V0077]
      (GO G0079)
L0078
      (RETURN V0070)))
```

4.3. MARKS

MARKS is the principal data structure involved in the Reachability Algorithm. At each iteration through the main loop it represents what is currently known about the reachability of each of the vertices in the graph:

MARKS[X] = "EXPLORED"
⇒ X is reachable and its successors have been noted as reachable
MARKS[X] = "BOUNDARY"
⇒ X is reachable and its successors have not been examined
MARKS[X] = "UNEXPLORED"
⇒ no path to X has yet been found

In the rest of this section, E, B, and U will denote "EXPLORED", "BOUNDARY", and "UNEXPLORED" respectively.

The abstract description for MARKS is as follows:

MARKS:
MAPPING (integers → {E, B, U})

Note that the computation of the inverse image of some range element is a common operation on MARKS. In such situations, it is often convenient to use an inverted representation. That is, rather than associating range elements with domain elements, sets of domain elements can be associated with range elements.

> *A mapping with typical domain element X and typical range element Y may be represented as a mapping with typical domain element Y and typical range element a collection with typical element X.*

MARKS_{inv}:
MAPPING ({E, B, U} → collections of integers)

At this point, the same two rules that were applied to SUCCESSORS can be applied to MARKS_{inv}:

> *A mapping may be represented explicitly.*
> *An explicit mapping may be stored in a single structure.*

MARKS_{inv}:
STORED MAPPING ({E, B, U} → collections of integers)

When selecting the structure in which to store the mapping, we may take advantage of the fact that the domain is a fixed set (E, B, and U):

> *A stored mapping whose domain is a fixed set of alternatives and whose typical range element is Y may be represented as a plex with one field for each alternative and with each field being Y.*

MARKS_{plex}:
PLEX (UNEXPLORED, BOUNDARY, EXPLORED)
EXPLORED: collection of integers
BOUNDARY: collection of integers
UNEXPLORED: collection of integers

A plex is an abstract record structure consisting of a fixed set of named fields, each with an associated substructure, but without any particular commitment to the way the fields are stored in the plex. In LISP, the obvious way to represent such a structure is with CONS cells:

MARKS_{lisp}:
CONS CELLS (UNEXPLORED BOUNDARY . EXPLORED)
EXPLORED: collection of integers
BOUNDARY: collection of integers
UNEXPLORED: collection of integers

Notice that we are now concerned with three separate collections which need not be represented the same way.

Since MARKS is inverted, the inverse image of an object under MARKS may be computed by retrieving the image of that object under MARKS_{inv}. If the domain object (e.g., B) is known at the time the program is constructed, this operation may be further refined into a simple retrieval of a field in the plex:

---

retrieve the BOUNDARY field of MARKS_{plex}

Likewise, the image of a domain object may be changed from one value to another (for example, from B to E) by moving the object from one collection to another:

---

remove X from the BOUNDARY field of MARKS_{plex};
add X to the EXPLORED field of MARKS_{plex}

4.4. BOUNDARY

BOUNDARY is the set of all vertices that map to B under MARKS. Since MARKS is inverted, this collection exists explicitly and a representation for it must be selected. The abstract description of BOUNDARY is as follows:

BOUNDARY:
COLLECTION (integer)

The operations that are applied to BOUNDARY include the addition and deletion of elements and the selection of some element from the collection. A linked list is often convenient for such operations. To derive a representation using cells

retrieve the element at location *L* of BOUNDARY$_{seq}$
L is any location

The next step is then to select the location to be used. The two most useful possibilities for sequential collections are the front and the back. Of these, the front is generally best for linked lists; although the back can also be used, it is usually less efficient:

If a location in a sequential collection is unconstrained, the front may be used.

retrieve the element at the front of BOUNDARY$_{list}$

The remaining steps are straightforward:

If a linked list is represented using linked free cells with a special header cell, the front location may be computed by retrieving the link from the first cell.

If linked free cells are implemented as a LISP list, the link from the first cell may be computed by using the function CDR.

If linked free cells are implemented as a LISP list, the element at a cell may be computed by using the function CAR.

The result of these three rule applications, when combined with the code for computing MARKS$_{inv}[B]$, is the following LISP code for computing "any element of MARKS^{-1}["BOUNDARY"]":

```
(CAR (CDR (CAR (CDR MARKSlisp))))
```

4.4.2. *Remove X from MARKS$_{inv}$["BOUNDARY"]*

Recall that one of the operations involved in changing the image of *X* from *B* to *E* is the removal of *X* from MARKS$_{inv}[B]$:

remove *X* from BOUNDARY

The first refinement step is similar to that of the "any element" operation:

If a collection is represented as a sequential collection, an element may be removed by removing the item at the location of the element in the collection.

remove the item at location *L* of BOUNDARY
L is the location of X

allocated from free storage, we apply a sequence of five rules, which lead from "collection" to "linked free cells" in the collection diagram given earlier:

A collection may be represented explicitly.

An explicit collection may be stored in a single structure.

A stored collection with typical element X may be represented as a sequential arrangement of locations in which instances of X are stored.

A sequential arrangement of locations with typical element X may be represented as a linked list with typical element X.

A linked list may be represented using linked tree cells.

BOUNDARY$_{cells}$:
LINKED FREE CELLS (integer)

It is often convenient to use a special header cell with such lists, so that the empty list need not be considered as a special case:

A special header cell may be used with linked free cells.

BOUNDARY$_{cells}$:
LINKED FREE CELLS (integer) *with special header cell*

Any use of cells allocated from free storage requires allocation and garbage collection mechanisms. In LISP, both are available with the use of CONS cells:

Linked free cells may be represented using a LISP list of CONS cells.

BOUNDARY$_{lisp}$:
LISP LIST (integer) *with special header cell*

4.4.1. *Any Element of MARKS^{-1}["BOUNDARY"]*

The main loop of the Reachability Algorithm is repeated until MARKS^{-1}["BOUNDARY"] (i.e., the BOUNDARY collection) is empty. At each iteration, one element is selected from the collection:

retrieve any element of BOUNDARY

The first refinement step for this operation depends on the earlier decision to represent BOUNDARY as a sequential collection:

If a collection is represented as a sequential collection, the retrieval of any element in the collection may be implemented as the retrieval of the element at any location in the collection.

Normally, determining the location of an element in a sequential collection involves some kind of search for that location. In this case, however, the location is already known, since X was determined by taking the element at the front of BOUNDARY:

If an element X was determined by retrieving the element at location L of a sequential collection C, then L is the location of X in C.

---

remove the item at the front of BOUNDARY

(Testing the condition of this rule involves tracing back over the steps that produce the particular element X and determining that, indeed, the location of X in BOUNDARY is the front.) From this point on, the program construction process is relatively straightforward, and similar to the "any element" derivation. The end result is the following LISP code:

```
(RPLACD (CAR (CDR MARKSlisp))
        (CDR (CDR (CAR (CDR MARKSlisp)))))
```

4.5. UNEXPLORED

The UNEXPLORED collection contains all of those vertices to which no path has yet been found. The initial description of UNEXPLORED is the same as that of BOUNDARY:

UNEXPLORED:
COLLECTION (integer)

The only operations applied to this collection are membership testing, addition, and deletion. For such operations, it is often convenient to use a different representation than simply storing the elements in a common structure (as was done with the BOUNDARY collection):

A collection may be represented as a mapping of objects to Boolean values; the default range object is FALSE.

UNEXPLORED$_{map}$:
MAPPING (integers → {TRUE, FALSE})

Having decided to use a Boolean mapping, all of the rules available for use with general mappings are applicable here. In particular, the same sequence of rules that was applied to derive the representation of SUCCESSORS can be applied with the following result:

UNEXPLORED$_{lisp}$:
LISP ARRAY ({TRUE, FALSE})

Thus, UNEXPLORED is represented as an array of Boolean values, where the entry for index k is TRUE if vertex k is in the UNEXPLORED collection and FALSE otherwise.

The implementation of the "change MARKS[Y] from U to B" operation involves removing Y from the UNEXPLORED collection. Since UNEXPLORED is represented differently from BOUNDARY, removing an element must also be done differently. In this case, four rules, together with the LISP representation of FALSE as NIL, give the following LISP code:

```
(SETA UNEXPLOREDlisp Y NIL)
```

4.6. Final program

The other aspects of the implementation of the Reachability Algorithm are similar to those we have seen. The following is a summary of the final program. (Here, $X[Y]$ denotes the Yth entry in the array X and X:Y denotes the Y field of the plex X.)

Reachability Program

```
VERTICES ← input a list of integers;
SUCCESSORSinput: ← input an association list of ⟨integer, list of
                                                    integers⟩ pairs;
SUCCESSORS ← create an array of size 100;
for all X in the list SUCCESSORSinput;
  SUCCESSORS[X:DOMAIN] ← X:RANGE;
START ← input an integer;
MARKS:UNEXPLORED ← create an array of size 100;
MARKS:BOUNDARY ← create an empty list with header cell;
MARKS:EXPLORED ← create an empty list with header cell;
for all X in the list VERTICES:
  MARKS:UNEXPLORED[X] ← TRUE;
MARKS:UNEXPLORED[START] ← FALSE;
insert START at front of MARKS:BOUNDARY;
loop:
  if MARKS:BOUNDARY is the empty list then exit;
  X ← front element of MARKS:BOUNDARY;
  insert X at front of MARKS:EXPLORED;
  remove front element of MARKS:BOUNDARY;
  for all Y in the list SUCCESSORS[X];
    if MARKS:UNEXPLORED[Y] then
      MARKS:UNEXPLORED[Y] ← FALSE;
      insert Y at front of MARKS:BOUNDARY;
  repeat;
output MARKS:EXPLORED.
```

5. Implementation

There are three important aspects to PECOS's implementation: the representation of program descriptions in a refinement sequence, the representation of programming rules, and the control structure. In this section, each of these will be considered briefly.

5.1. Representation of Program Descriptions

Each program description in a refinement sequence is represented as a collection of nodes, with each node labelled by a particular programming concept. For example, a node labelled IS-ELEMENT represents an operation testing whether a particular item is in a particular collection. Each node also has a set of links or properties related to the node's concept; for example, an IS-ELEMENT node has properties (named ELEMENT and COLLECTION) for its arguments. Although property values may be arbitrary expressions, they are usually links to other nodes (as in the IS-ELEMENT case). Fig. 3 shows part of the internal representation for the expression IS-ELEMENT(X,INVERSE(Y,Z)).[10] Each box is a node. The labels inside the boxes are the concepts and the labeled arrows are property links. The argument links of an operation all point to other operations.

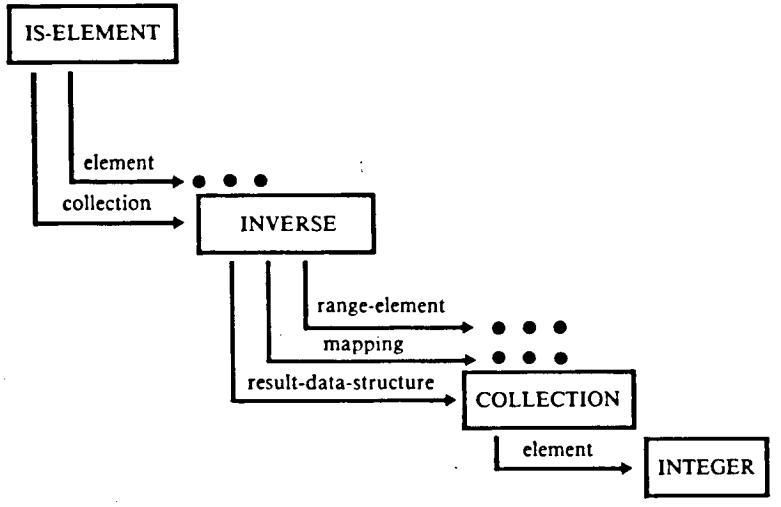


FIG. 3.

For example, the COLLECTION operand of the IS-ELEMENT node is the INVERSE node. The value computed by an operation is indicated by the RESULT-DATA-STRUCTURE property. Thus, the COLLECTION node represents the data structure passed from the INVERSE operation to the IS-ELEMENT operation. (Note that this collection is only implicit in the corresponding English description.) Since refinement rules for an operation have conditions both on the data structure it produces and on the data structures produced by its operands, the refinement of this COLLECTION node enables the refinements of the IS-ELEMENT and INVERSE nodes to be coordinated. Thus, for example, the refined inverse operation does not produce a linked list when the refined membership test expects a Boolean mapping. This explicit representation of every data structure passed from one operation to another is perhaps the most important feature of PECOS's representation of program descriptions.

[10] Read "Is X an element of the inverse image of Y under the mapping Z?".

5.2. Representation of Programming Rules

PECOS's rules all have the form of condition–action pairs, where the conditions are patterns to be matched against subparts of descriptions, and the actions are particular modifications that can be made to program descriptions. Based on the action, the rules are classified into three types:

Refinement rules refine one node pattern into another. The refined node is typically created at the time of rule application. These are the rules which carry out the bulk of the refinement process, and are by far the most common type.

Property rules attach a new property to an already existing node. Property rules are often used to indicate explicit decisions which guide the refinements of distinct but conceptually linked nodes.

Query rules are used to answer queries about a particular description. Such rules are normally called as part of the process of determining the applicability of other rules.

The internal representation of the rules is based on this classification:

(REFINE ⟨*node pattern*⟩⟨*refinement node*⟩)
(PROPERTY ⟨*property name*⟩ ⟨*node pattern*⟩ ⟨*property value*⟩)
(QUERY ⟨*query pattern*⟩ ⟨*query answer*⟩)

where REFINE, PROPERTY, and QUERY are tags indicating rule type. In the REFINE and PROPERTY rules, each ⟨*node pattern*⟩ consists of a ⟨*concept*⟩ and an ⟨*applicability pattern*⟩. For example, the following rule

> *A sequential arrangement of locations with typical element X may be represented as a linked list with typical element X.*

is represented as a refinement rule:

```
(REFINE (SEQUENTIAL-COLLECTION
                (GET-PROPERTY ELEMENT (BIND X)))
        (NEW-NODE LINKED-LIST
                (SET-PROPERTY ELEMENT X)))
```

SEQUENTIAL-COLLECTION is the ⟨*concept*⟩ and (GET-PROPERTY ELEMENT (BIND X)) is the ⟨*applicability pattern*⟩.

Several other issues have arisen in the design of the rule base organization, including an indexing scheme for efficient rule retrieval, the design of a pattern matcher, and the breakdown of the condition patterns into separate parts for different uses. Detailed discussions of these issues may be found elsewhere [3]. It should also be noted that several kinds of rules are not easily expressed in the current formalism. For example, more general inferential rules (such as the test constructor used by Manna and Waldinger [23]) and rules about certain kinds of data flow (such as merging an enumeration over a set with the enumeration that constructed it) can not be described conveniently with the current set of rule types. It is not clear how difficult it would be to extend PECOS to handle such cases.

5.3. Control Structure

PECOS uses a relatively simple agenda control structure to develop refinement sequences for a given specification: in each cycle, a task is selected and a rule applied to the task. While working on a given task, subtasks may be generated; these are added to the agenda and considered before the original task is reconsidered. There are three types of tasks:

- (REFINE *n*) specifies that node *n* is to be refined.
- (PROPERTY *p n*) specifies that property *p* of node *n* is to be determined.
- (QUERY rel arg1 arg2 · · ·) specifies that the query (rel arg1 arg2 · · ·) must be answered.

When working on a task, relevant rules are retrieved and tested for applicability. For example, if the task is (REFINE 72) and node 72 is an IS-ELEMENT node, then all rules of the form (REFINE (IS-ELEMENT · · ·) · · ·) will be considered. When testing applicability, it may be necessary to perform a subtask. For example, an argument may need to be refined in order to determine if it is represented as a linked list. This is, in fact, quite common: the refinement of one node is often the critical factor making several rules inapplicable to a task involving another node.

When several rules are applicable, each would result in a different implementation. When a single rule cannot be selected (either by the choice-making heuristics the user, or LIBRA), a refinement sequence can be split, with each rule applied in a different branch. As a result, PECOS creates a tree of descriptions in which the root node is the original specification, each leaf is a program in the target language, and each path from the root to a leaf is a refinement sequence. With the current knowledge base, most refinement sequences lead to complete programs. For the few that do not, the cause is generally that certain operation/data structure combinations do not have any refinement rules. For example, there are no rules for computing the inverse of a distributed mapping, since this might require enumerating a very large set, such as the set of atoms in a LISP core image. If PECOS encounters a situation in which no rules are applicable, the refinement sequence is abandoned.

Further details of the control structure and the context mechanism used for the tree of descriptions may be found elsewhere [3].

6. Discussion

6.1. A Search space of correct programs

The problem of choosing between alternative implementations for the same abstract algorithm is quite important, since the efficiency of the final program can vary considerably with different implementation techniques. Within the framework of Ψ's synthesis phase, this problem has been broken into two components:

(1) constructing a search space whose nodes are implementations (possibly only partial) of the abstract algorithm, and
(2) exploring this space, making choices on the basis of the relative efficiency of the alternatives.

The first is provided by PECOS's rules and refinement paradigm; the second is provided by LIBRA [17], Ψ's Efficiency Expert. PECOS can thus be viewed as a "plausible move generator" for which LIBRA is an "evaluation function".

6.1.1. Refinement trees

The space of alternative implementations generated by PECOS can be seen as a generalization of refinement sequences. Whenever alternative rules can be applied (and hence, alternative implementations produced), a refinement sequence can be split. Thus, we have a refinement tree, as illustrated in Fig. 4. The root of such a tree is the original specification, the leaves are alternative implementations, and each path is a refinement sequence.

Experience, both with PECOS alone and together with LIBRA, has shown that a refinement tree constitutes a fairly "convenient" search space. First, the nodes (program descriptions) all represent "correct" programs.[11] Each node represents a step in a path from the abstract specification to some concrete implementation of it. When paths cannot be completed (as happens occasionally), the cause is generally the absence of rules for dealing with a particular program description, rather than any inherent problem with the description itself. Second, the refinement paradigm provides a sense of direction for the process. Alternatives at a choice point represent reasonable and useful steps toward an implementation. Third, the extensive use of intermediate level abstractions makes the individual refinement steps fairly small and "understandable". For example, the efficiency transformations associated with the rules for use by LIBRA (see Section 6.1.3) are simpler than they would be if the intermediate levels were skipped.

[11] Assuming correctness of the rules, of course; see Section 6.3.

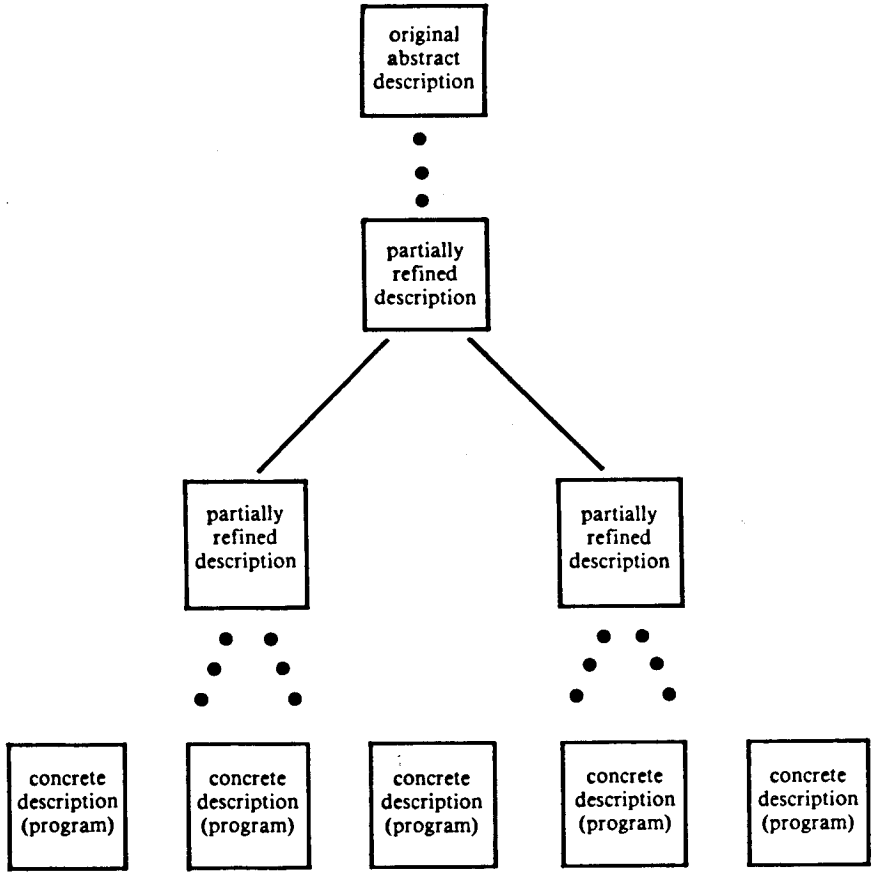


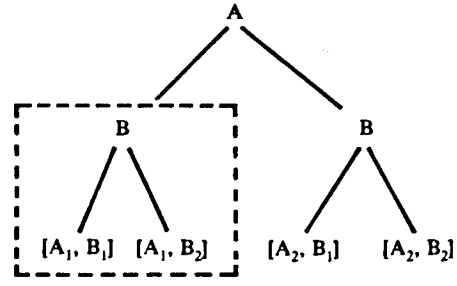
FIG. 4.

6.1.2. *Techniques for reducing the space*

The size of refinement trees (i.e., the total number of rule applications) may be reduced without eliminating any of the alternative implementations by taking advantage of the fact that many of the steps in a refinement sequence may be reordered without affecting the final implementation: the only absolute ordering requirement is that one task must be achieved first if it is a subtask of another. For example, if a program involves two collections, the refinement steps for each must occur in order, but the two subsequences may be intermingled arbitrarily.

PECOS postpones consideration of all choice points until the only other tasks are those for which some choice point is a subtask.[12] As a result, refinement trees tend to be "skinny" at the top and "bushy" at the bottom. Experience has shown that a considerable reduction in tree size can result. For example, the size of the tree for the four implementations of CLASS (see Section 3) was reduced by about one third by using this technique.

When two choice points are sufficiently independent that a choice for one may be made regardless of what choice is made for the other, a simple extension of the choice point postponement technique permits further pruning. For example, suppose there are two choice points (A and B), each having two applicable rules. The four leaves of the entire tree represent the cross-product of the alternatives for each of the choices. If one (say A) is considered first, then several further refinement steps (for which A had been a subtask) may be made before B needs to be considered. Since A is independent of B, the two paths for A can be carried far enough that a preference for one path over the other can be determined before B is considered. Thus, the alternatives for B along the other path need not be considered at all. In the tree shown below, the branches inside the box need not be explored if A_2 can be selected over A_1 independently of what choice is made for B.



6.1.3. *Techniques for making choices*

When PECOS is running alone, choices between alternative rules are made either by the user or by a set of about a dozen choice-making heuristics. Some of these heuristics are intended to prune branches that will lead to dead ends (situations in which no rules are applicable). For example, PECOS has no rules for adding an element at the back of a linked list. One of the heuristics (for selecting a position at which an element should be added to a sequential collection) tests whether the sequential collection has been refined into a linked list, and if so rejects "back" as a possibility. One interesting feature of such heuristics is that they embody knowledge about the capabilities of the system itself, and thus should be changed as rules are added and the system's capabilities change.

Other heuristics deal with decisions that can be made on a purely "local" basis, considering only the node being refined and the alternative rules. Sometimes one alternative is known a priori to be better than another; if both are applicable the better alternative should be taken. For example, one of PECOS's heuristics

[12] With the current rule set, choice points are relatively infrequent. In the Reachability Program, for example, there were about three dozen choice points in a refinement sequence that involved about one thousand rule applications.

prefers PROGN to PROG constructs: (PROGN ···) is better than (PROG NIL ···). In other cases, the cost difference between two alternatives isn't very great, but one is more convenient for most purposes. For example, PECOS has a heuristic that suggests always using special headers for linked free cells, since the extra cost is low (one extra cell) and they are often more easily manipulated.[13]

Experience with programs such as those in Section 3 suggests that PECOS's heuristics are capable of handling a majority of the choice points that arise in refinement sequences. About two-thirds of the three dozen choice points involved in developing the Reachability Program were handled by the heuristics. For example, the front/back heuristic for linked lists handled the selection of the front position of BOUNDARY for the "any" operation. On the other hand, the selection of the array representation for SUCCESSORS was beyond the capabilities of the heuristics, since it involved global considerations, such as the size of the set of integers that might be nodes, the density of nodes within that set, and the cost of conversion.

When PECOS and LIBRA are used together, LIBRA provides the search strategy and makes all choices. LIBRA's search strategy includes the choice point postponement technique described earlier, as well as techniques for identifying critical choice points and allocating resources. LIBRA's choice-making techniques include local heuristics, such as PECOS's, as well as global heuristics. When the heuristics are insufficient, or when a choice point is determined to be especially important, analytic methods are applied. Using cost estimates for the intermediate level constructs, LIBRA computes upper and lower bounds on the efficiency of all refinements of a node in the refinement tree. Then standard search techniques (such as branch and bound) can be used to prune the tree. The efficiency estimates also rely on specific information about the user's algorithm, such as set sizes and branching probabilities. These are provided as part of the original abstract description, and are computed for refinements by efficiency transformations associated with PECOS's refinement rules. PECOS and LIBRA's behavior together has been fairly good. For example, they constructed the linear implementation of the prime number algorithm given by Knuth. The reader is referred elsewhere for further details of LIBRA's operation [17], and for a detailed example of PECOS and LIBRA operating together [4].

6.2. Rule generality

One of the critical issues involved in this knowledge-based approach to automatic programming is the question of rule generality: will many (or even some) of PECOS's rules be useful when other programming domains are codified? If so, there is some hope that a generally useful set of rules can eventually be developed. If not, then the knowledge base required of such a system may be prohibitively large. While it is too early to give a definitive answer, there are several encouraging signs. First, as described in Section 3, PECOS has successfully implemented programs in a variety of domains, ranging from simple symbolic programming to graph theory and even elementary number theory. As mentioned earlier, very few additional rules (about a dozen out of the full 400) needed to be added to the knowledge base in order to write the reachability and prime number programs. Second, the fact that LISP-specific rules could be replaced by SAIL-specific rules, leaving most of the knowledge base unchanged, also suggests a degree of generality in the rules.

In the long run, the question of rule generality can only be resolved by trying to extend the rule set to cover new domains. Work is underway at Yale to codify the programming knowledge needed for elementary graph algorithms [2]. The early signs again suggest the utility of knowledge about sets and mappings. For example, the notion of an enumeration state seems important for enumerating the nodes in a graph, just as it is important for enumerating the elements of a set. The MARKS mapping used in the abstract algorithm in Section 4 encodes the state of the enumeration of the reachable nodes in the graph. As another example, consider the common technique of representing a graph as an adjacency matrix. In order to construct such a representation, only one rule about graphs need be known:

> *A graph may be represented as a pair of sets: a set of vertices (whose elements are primitive objects) and a set of edges (whose elements are pairs of vertices).*

The rest of the necessary knowledge is concerned with sets and mappings and is independent of its application to graphs. For example, in order to derive the bounds on the matrix, one need only know that primitive objects may be represented as integers, that a set of otherwise unconstrained integers may be represented as a sequence of consecutive integers, and that a sequence of consecutive integers may be represented as lower and upper bounds. To derive the representation of the matrix itself, one need only know PECOS's rules about Boolean mappings and association tables, plus the fact that a table whose keys are pairs of integers in fixed ranges may be represented as a two-dimensional matrix.

6.3. Issues of correctness

It seems plausible that access to a large base of programming knowledge would help reduce the search involved in program verification. Consider, for example, the problem of determining the loop invariant in the following program for adding 1 to every entry in an array A:

[13] Note that the heuristic doesn't take into account whether or not the extra cell actually helps in the particular case under consideration. And, indeed, it may be difficult to tell at the time the choice is made.

```
(PROG (I)
      (SETQ I 1)
RPT   (COND ((IGREATERP I N)
             (RETURN NIL)))
      (SETA A I (IPLUS (ELT A I) 1))
      (SETQ I (IPLUS I 1))
      (GO RPT))
```

The invariant, to be attached "between" the RPT label and the COND expression, is as follows:

$$\forall k(1 \leqslant k < I \supset A[k] = A_0[k]+1) \wedge (I \leqslant k < N \supset A[k] = A_0[k])$$

where $A[k]$ denotes the current kth entry of A and $A_0[k]$ denotes the initial kth entry. Knowing the sequence of rule applications that produced this code gives us the invariant almost immediately. The $\forall k$ is suggested by the fact that the loop implements an enumeration over a sequential collection represented as an array subregion. Since the enumeration order is the stored order, and the enumeration state is held by I, we know that the action should have been done for all and only those indices less than I. And since the enumeration is total and the bounds on the subregion are 1 and N, we know the indices of concern are those between 1 and N.

But on another level, we might ask whether PECOS's rules are themselves correct. For refinement rules such as these, a reasonable definition of correctness might be: a rule is correct if, for all programs to which the rule may be applied, all relevant properties of the program are preserved under rule application. Thus, if the original abstract algorithm is correct (i.e., has the properties desired by the user), and if all rules that are applied are correct in this sense, then the final implementation is a correct one. What properties are the relevant ones? Clearly, the same kinds of properties that have been considered in the traditional approaches to program verification: that the value returned by a refined operation be the same as the value returned by the abstract operation under the same conditions, and that the side effects of a refined operation be the same as those of the abstract operation. But note that the question of side effects is complicated by the fact that some of the side effects at a refined level cannot even be discussed at an abstract level. For example, the fact that list cells are being modified when a new object is added to a list doesn't have any relevance at the more abstract level of adding an object to a sequential collection.

Although it would clearly be useful, no formal proofs of PECOS's rules have yet been made. In fact, the relevant properties of the abstract operators have only been specified informally. The goal of this research was rather to lay the groundwork by identifying the useful concepts and relationships, as a necessary precursor to a complete formalization. Nonetheless, some of the issues mentioned above have come up in the process of developing a set of rules that seems at least informally to be correct. As a simple example, consider the operation of inserting a new cell after a given cell in a linked list. PECOS's representation of this operation is approximately (INSERT-AFTER-CELL ⟨cell⟩ ⟨new-object⟩). The obvious refinement rule here indicates that a new cell should be created (for ⟨new-object⟩ and the link from ⟨cell⟩) and that a link to this new cell should be stored in the link field of ⟨cell⟩. Implementing this as a simple macro expansion gives roughly (REPLACE-LINK ⟨cell⟩ (CREATE-CELL ⟨new-object⟩ (GET-LINK ⟨cell⟩))). But notice that, if ⟨cell⟩ is an expression (possibly with side effects), rather than simply the value of a variable, the expression is evaluated twice, which might then violate the correctness criteria discussed above, or at least be somewhat inefficient. Thus, the correct refinement involves storing the value of ⟨cell⟩ before performing the above operations.

7. Approaches to Automatic Programming

The term "automatic programming" is rather difficult to define. In part the problem lies in the long history of the term: it was used over twenty years ago in discussions about early programming languages [1]. Nonetheless, for the sake of this discussion, let us define automatic programming rather loosely as an attempt to automate some part of the programming process. A variety of approaches have been taken in facing this problem, including extending language development to higher level languages [21, 25], formalizing the semantics of programming languages [5, 9], and many attempts involving the application of artificial intelligence techniques [8, 12, 27, 28, 31]. PECOS is an attempt to apply yet another artificial intelligence paradigm, knowledge engineering, to the same goal of automatic programming.

While it is too early to tell how far any of these approaches will lead, insight may be gained by comparing them. In the rest of this section, we will consider four systems, each of which exemplifies a different approach, and compare them with respect to certain fundamental issues of automatic programming. The four systems are DEDALUS [23], based on a *deductive* approach; Darlington and Burstall's system [8], involving a *transformational* approach; Low's system [21], based on *high level languages*; and PECOS, based on a *knowledge-based* approach. Of course, there are aspects of each approach in each of the systems, and, as will become clearer, the differences are less than they might seem at first. But nonetheless, each system will serve to illustrate one approach.

7.1. Summaries of three other automatic programming systems

Before comparing these systems, let us briefly review DEDALUS, Darlington and Burstall's system, and Low's system.

Although DEDALUS employs transformation rules extensively, it is classified as an example of the deductive approach to automatic programming because of the use of deduction in developing the control structure of the target program.

Briefly, DEDALUS takes a program specified non-algorithmically in terms of sets (and lists) of integers, and tries to apply various transformations that introduce primitive constructs in the target language. In the process, conditionals and recursive calls may be introduced when their need is recognized by the deductive system. For example, consider implementing a program to compute $x <$ all(l), that is, whether x is less than every member of l, where x is an integer and l is a list of integers. There are two transformation rules for the all construct. The first transforms P(all(l)) into true if l is the empty list; the second transforms P(all(l)) into P(head(l)) and P(all(tail(l))), if l is not empty. Since l cannot be proved to be either empty or not empty, the original construct is transformed into if empty(l) then true else ($x <$ head(l)) and ($x <$ all(tail(l))). DEDALUS then notices that $x <$ all(tail(l)) is an instance of the original goal $x <$ all(l), suggesting the use of a recursive call to the function being defined. In this way, various control constructs can be introduced into the program by the deductive mechanism. Among the programs which DEDALUS has successfully constructed are programs for computing the maximum element of a list, greatest common divisor, and cartesian product.

Darlington and Burstall developed a system (which we will refer to as DBS) for improving programs by a transformation process. Programs are specified in the form of recursive definitions in the domain of finite sets, and are transformed into an imperative language which includes constructs such as "while" and assignment, as well as recursion. The transformation process goes through four stages: recursion removal, elimination of redundant computation, replacement of procedure calls by their bodies, and causing the program to reuse data cells when they are no longer needed. In addition, DBS could implement the set operations with either bitstring or list representations. Among the programs optimized by DBS are an in-place reverse from its recursive definition and an iterative Fibonacci from its doubly recursive definition. In a later system [6], they incorporated various strategies for developing recursive programs (e.g., a "folding" rule which is similar to Manna and Waldinger's recursion introduction rules).

Low has developed a system (which will be referred to here as LS) for automatically selecting representations for sets. LS accepts programs specified in a subset of SAIL which includes lists and sets as data types, and uses a table of concrete (at the machine code level) implementations for these data types and the associated operations, together with information about the efficiency of the implementations. The available implementation techniques include sorted lists and arrays, balanced trees, bitstrings, hash tables, and attribute bits. LS first partitions the various instances of sets and lists into groups that should be implemented uniformly, then analyzes the frequency of use of the various constructs, both statically and dynamically with sample data, and finally uses a hill-climbing technique to find an optimal implementation. LS has been applied to such programs as insertion sort, merge sort, and transitive closure.

7.2. Techniques of program specification

Programs are specified to the four systems in basically two ways: DEDALUS accepts specifications such as the following:

compute $x <$ all(l)
  where x is a number
  and l is a list of numbers

This can be seen as simply another way of giving input and output predicates, where the input predicate is the "where" clause and the output predicate is the "compute" clause. The other three systems all accept specifications as abstract algorithms in terms of set operations; PECOS also allows operations on mappings. However, the difference is perhaps less than it might seem, since an abstract operator is simply a shorthand for some implicit input and output predicates. For example, inverse(x, y) is simply another way of saying the following (assuming the domain and range of y are integers):

compute $\{z \mid y(z) = x\}$
  where y is a mapping from integers to integers
  and x is an integer

The main advantage of the input/output form over the abstract algorithm is increased generality: there may not be a concept whose implied input/output predicates are precisely those desired. On the other hand, the algorithmic form often seems more natural.

There are other specification techniques that one could imagine, such as example input/output pairs, natural language, and even dialogue. In the long run, it seems that a mixture of all of these techniques (and probably more) will be needed—perhaps the prime criterion should be the "naturalness" of the technique to the person specifying the program and the domain under consideration.

7.3. Side effects

The problem of side effects has long been a difficult one for problem-solving and automatic programming systems. HACKER [28], for example, was unable to solve certain kinds of robot problems because the side effects of operations to achieve one goal might interfere with previously achieved goals. Consequently, many automatic programming systems deal with side effects in only limited ways. DBS, for example, only accepts programs specified in an applicative (i.e., without side effects) language, although assignments to variables are allowed in the target language. In addition, the final stage of transformation involves attempting to reuse discarded list cells, instead of calls to CONS. DEDALUS allows only a few target language operations that have side effects (basically, different kinds of assignment), along with specification language constructs like "only z changed",

meaning that z is the only variable that may be modified. The problem of interfering goals is faced by labelling certain goals as "protected", and requiring that those which cannot be proved to hold must be re-achieved. LS and PECOS l oth allow a richer set of specification operators with side effects, such as adding to and removing elements from sets (and, in PECOS's case, operators such as changing the image of an object under a mapping). In each case, the knowledge base (i.e., LS's implementation table and PECOS's rules) include techniques for implementing the specific operators. As mentioned earlier, the nature of side effects sometimes requires that PECOS's rules not be implemented as simple macro expansion: the rule for inserting an object into a list requires that the insertion location be saved, since a call to (REPLACE-LINK ⟨cell⟩ (CREATE-CELL ⟨new-object⟩ (GET-LINK ⟨cell⟩))) would involve computing ⟨cell⟩ twice. In any case, it seems that the problem of side effects is likely to plague automatic programming systems for some time. The problem is, unfortunately, unavoidable, since efficient implementations often require explicit and deliberate use of side effects.

7.4. Abstraction and refinement

Just as abstraction and refinement have become increasingly important in programming methodology [7], and in recent developments in programming languages [20, 33], they have also begun to play a role in automatic programming. In DEDALUS, the role is relatively minor: some of the specification language constructs can be viewed as abstractions. For example, the all construct, as in $x <$ all(l), is essentially the same as PECOS's abstract control structure FOR-ALL-TRUE, as in (FOR-ALL-TRUE y l (LESS x y)). Abstraction and refinement are much more important in DBS, LS, and PECOS. Each accepts programs specified in abstract terms involving sets (and, in PECOS's case, mappings) and produces a concrete program which is essentially a refinement of the abstract program. For DBS and PECOS the target language is LISP, and for LS the target language is PDP-10 machine code. Perhaps the greatest distinction between the systems is PECOS's use of multiple levels of abstraction. While DBS and LS both refine in one step from the abstract concepts to the target language constructs, PECOS may go through as many as a half dozen levels between the specification level and the target level. This seems to offer several advantages. First, multiple levels contribute to an economy of knowledge. The example noted earlier, sequential collections, illustrates this well: much of the knowledge about linked lists and arrays is common to both and can be stored once as knowledge about sequential collections, rather than once for linked lists and once for arrays (and once for every other kind of sequential collection). Second, multiple levels facilitate the use of different representations in different situations. This was a major problem with LS: the sets to be represented were partitioned into equivalence classes such that all arguments of any operator were in the same equivalence class. The effect, in the case of a transitive closure program, was that all of the sets were represented in the same way. The motivation for this partitioning was that the tables would have to be prohibitively large to allow each of the operators to take different kinds of arguments or to include representation conversions between them. In PECOS, the intermediate levels of abstraction provide convenient hooks for knowledge about converting between representations. For example, the BOUNDARY and UNEXPLORED collections of the Reachability Program are represented differently, while in LS's implementation of transitive closure the corresponding sets were forced to have the same representation. In fact, not only can PECOS avoid the requirement that several related data structures have the same representation, but the same data structure can even have different representations. For example, the SUCCESSORS mapping was input as an association list but converted into a Boolean array, which was more efficient in the inner loop. A third benefit of the muliple levels of abstraction, and perhaps this is the cause of the benefits just described, is that there is room for a rich interplay between the programming concepts involved. The single rule about representing collections as Boolean mappings, for example, leads to a variety of different collection representations because of the knowledge already known about mappings.

7.5. Dealing with alternative implementations

The ultimate goal of an automatic program synthesis system is to produce the best (or at least an adequate) target language program that satisfies the user's specifications. Thus, in a sense, the problem involves search in the space of all legal target programs. This space is obviously too large to be explored exhaustively, so all automatic synthesis systems incorporate (at least implicitly) some way of reducing this space. But notice that there are two aspects of the desired program: it must satisfy the user's requirements, and it should be the best such satisfactory program. Earlier systems (e.g., the Heuristic Compiler [27] and the theorem-prover based systems [12, 31]) basically faced only the first aspect: they were concerned with finding any program that worked. DEDALUS shares this concern: its goal is to find some program in the target language that satisfies the user's specifications. The only sense in which it faces the second aspect is that the user may disable certain rules in the hope that the program found using the remaining rules (if any is found) will be better. DBS, LS, and PECOS all focus on the second aspect, in that there is an explicit space of alternative implementations. DBS allows the user to choose between two alternative implementations of the basic set constructs, lists and bitstrings. LS has its table of seven alternative implementations for set operations, and chooses among them automatically. PECOS's rules deal with about a dozen representations for collections (although not as varied as LS's representations), about a half dozen representations for mappings, and essentially two different enumeration techniques (ordered and unordered). There are several questions that one may ask about such a space of alternative implementations:

Does it include the desired implementation? Are there any dead-end or incorrect paths? How easy is it to explore? LS includes implementation techniques that PECOS does not (e.g., AVL trees). On the other hand, PECOS includes global considerations that LS does not. As noted above, PECOS avoids the restriction that all operands to a set operator must be represented in the same way, and even allows the same set to be represented in different ways in different places in the program. Thus, both LS's and PECOS's spaces include a variety of implementations for a given abstract algorithm, but the spaces seem to differ along slightly different dimensions. With respect to dead-end or incorrect paths, LS clearly has none. As discussed earlier (see Section 6.1), PECOS occasionally has a few dead-ends but no incorrect paths. How easy are the spaces to explore? In LS, after the partitioning restriction is made, the space is searched with a hill-climbing technique that seems both natural and convenient, although the possibility remains that the optimal program may be missed. As discussed in Section 6.1, PECOS's use of intermediate-level abstractions seems to facilitate greatly the process of making choices. PECOS has about a dozen choice-making heuristics that seem able to handle about two-thirds of the cases that arise in practice. LIBRA, Ψ's Efficiency Expert, uses more analytic methods, again taking advantage of the intermediate-level abstractions. In several test cases, LIBRA has performed quite well, although space limitations have prevented PECOS and LIBRA from being applied together to programs as large as the Reachability Algorithm. In the long run, it seems that the problem of choosing among alternative implementations will grow in importance for automatic programming systems, and that better techniques will be required, both for generating spaces that include the desired target programs and are convenient for exploration, and for choosing from among alternatives in such spaces. As far as the choice-making process is concerned, it seems a good guess that heuristic methods will become increasingly important, and that analytic methods (largely because of the cost of using them) will be reserved for cases in which the heuristics are inapplicable.

7.6. The role of deduction

In the earliest attempts to apply AI techniques to automatic programming, deduction (that is, the use of some kind of relatively general purpose theorem prover or problem solver) played a central role. The Heuristic Compiler was based on problem solving within the GPS framework [27]. The work of Green [12] and Waldinger [31] both involved extracting a program directly from a proof of a predicate calculus theorem derived from the input/output specifications of the program. In the more recent systems discussed here, the role of deduction seems less central. In DEDALUS, deduction is used for three purposes. First, some of the transformation rules have conditions associated with them, and these rules cannot be applied unless the conditions have been proven true in the current context. Second, in cases where the conditions have not been proven, a conditional control structure may be introduced in order to provide contexts in which the conditions are provable (and hence, the rule can be applied). Finally, in trying to prove or disprove "protected" conditions, deduction plays an important role in DEDALUS's handling of side effects. In DBS, a simple equality-based theorem prover plays an auxiliary role, similar to the first use of deduction in DEDALUS. Each recursive-to-iterative transformation rule has an associated set of equations over the primitives (e.g., equations satisfied only by associative operators); the rule can only be applied if the equations are satisfied. Deduction plays no role at all in LS. In PECOS, deduction plays a role only in a very limited sense: the QUERY subtasks of a task can be viewed as conditions in the DEDALUS sense described above, and the QUERY rules can be viewed as specialized deduction rules for handling particular situations (as opposed to giving the condition to a more general theorem prover).[14] In the Reachability Program, for example, the "deduction" that the front of BOUNDARY is the location of X was handled by QUERY rules. In the long run, it seems clear that the knowledge-based approach will require access to a more general deductive mechanism than simply a set of QUERY rules. First, it will be impossible to put in rules for all of the kinds of conditions that will need to be tested. For example, consider the following rule:

> *If it is known that an object is larger than all the elements of an ordered sequence, then the object may be added to the sequence by inserting it at the back.*

The "it is known that . . ." should clearly be tested by calling a deductive mechanism. And second, it will also be impossible to put in refinement and transformation rules to handle all possible cases that may arise in program specifications, and a general mechanism may provide the backup capability to handle the extra cases. Nonetheless, the appropriate role for a general deductive mechanism seems to be as an adjunct to the synthesis process, rather than as the driving force behind it.

7.7. Generality vs. power

It has often been stated (e.g., see [10, 11, 19]) that there is a trade-off between generality and power: techniques that are general will not help much in specific complex situations, and techniques that are powerful in particular complex situations will not be applicable in very many. We can see the same trade-off in the four automatic programming systems being discussed here. DEDALUS seems to occupy a point near the "general" end of the spectrum: It is designed to apply a general deductive framework to a wide variety of different problems specified with input/output predicates, but has not yet been successfully applied to very

[14] In fact, the transformation rules of DEDALUS, LS, and PECOS can all be seen as specialized (or perhaps even "compiled") deduction rules; and under this view the control structures serve as special-purpose deductive mechanisms.

complex programs. LS seems to be at the "power" end of the spectrum: it does extremely well at selecting data structure representations for sets, but is inapplicable to other programming problems; even to enable it to choose representations for a machine other than a PDP-10 would require redoing the tables completely. Yet even in these two cases, one can see traces of the other end of the spectrum. DEDALUS includes a large number of specific transformations that increase its power, and LS's control structure, including the partitioning and hill-climbing, could clearly be applied to other situations. DBS and PECOS seem to occupy the middle ground of the spectrum. While DBS's transformations are not universally applicable, they are certainly more general than those of LS. And while PECOS's rules are relatively specific to collections and mappings (and fairly powerful when they can be applied), they seem to possess a degree of generality, as suggested by the different domains in which they have been successfully applied (see Section 3). But note that this "generality" is concerned with a set of rules, rather than with individual rules. This does not necessarily mean that generality can be achieved by incorporating larger and larger numbers of specific rules, but my own belief about the future of automatic programming systems is that, in order to be useful and powerful in a variety of situations, they must necessarily incorporate a large number of rather specific detailed rules (or facts, or frames, or . . .), together with fairly general mechanisms (deductive, analytic, . . .) that can handle the situations in which the rules are inapplicable. And I would suggest that the organization will not be one driven by the general mechanism, with guidance from the rules, but rather one driven by the rules, with the general mechanisms for problem cases.

8. Assessment

The development of PECOS represents the final stage in an experiment investigating a knowledge-based approach to automatic programming. The essence of this approach involves the identification of concepts and decisions involved in the programming process and their codification into individual rules, each dealing with some particular detail of some programming technique. These rules are then represented in a form suitable for use by an automatic programming system.

As seen in Section 4, the process of constructing an implementation for an abstract algorithm involves considering a large number of details. It seems a reasonable conjecture that some kind of ability to reason at a very detailed level will be required if a system is to "understand" what it is doing well enough to perform the complex tasks that will be required of future automatic programming systems. PECOS's ability to deal successfully with such details is based largely on its access to a large store of programming knowledge. Several aspects of PECOS's representation scheme contribute to this ability. The refinement paradigm has proved convenient for coping with some of the complexity and variability that seem inevitable in real-world programs. The use of several levels of abstraction seems particularly important.

One of the critical issues involved in the knowledge-based approach to automatic programming is the question of rule generality: will many (or even some) of PECOS's rules be useful when other programming domains (e.g., graph algorithms) are codified? A definitive answer to this question must wait for other domains to be codified, but PECOS's successful application to the varied algorithms described in Section 3 is an encouraging sign.

In the long run, perhaps the greatest benefit of the knowledge-based approach lies in the rules themselves. Most knowledge about programming is available only informally, couched in unstated assumptions. While such knowledge is usually understandable by people, it lacks the detail necessary for use by a machine. For part of the domain of elementary symbolic programming, PECOS's rules fill in much of the detail and many of the unstated assumptions. Taken together, the rules form a coherent body of knowledge that imposes a structure and taxonomy on part of the programming process.

ACKNOWLEDGEMENTS

Cordell Green, my thesis adviser, and the other members of the Ψ project have been a source of motivation and focus for this work. Interaction with Elaine Kant and her work on LIBRA has been especially beneficial. Juan Ludlow's development of rules for SAIL exposed some of the hidden assumptions in my rules. Brian McCune contributed greatly to the design of the specification language. Randy Davis, Jorge Phillips, Drew McDermott, and Richard Waldinger have provided very helpful comments on drafts of this paper. The referees comments were also quite valuable.

REFERENCES

1. Automatic coding: *Proceedings of a Symposium at the Franklin Institute*, Philadelphia, Pennsylvania, April 1957.
2. Barstow, D. R., *Codification of programming knowledge: graph algorithms* (Yale University, Department of Computer Science, TR 149, December 1978).
3. Barstow, D. R., *Knowledge-based Program Construction* (Elsevier North-Holland, New York, 1979).
4. Barstow, D. R. and Kant, E., 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, 19–31.
5. Buchanan, J. and Luckham, D., *On automating the construction of programs* (Stanford University, Computer Science Department, AIM-236, May 1974).
6. Burstall, R. M. and Darlington, J., A transformation system for developing recursive programs. *Journal of the ACM* **24** (January 1977) 44–67.
7. Dahl, O.-J., Dijkstra, E. W. and Hoare, C. A. R., *Structured Programming* (Academic Press, New York, 1972).
8. Darlington, J. and Burstall, R. M., A system which automatically improves programs. *Acta Informatica* **6** (1976) 41–60.
9. Dershowitz, N. and Manna, Z., The evolution of programs: a system for automatic program modification. *IEEE Transactions on Software Engineering* (November 1977) 377–385.
10. Feigenbaum, E. A., The art of artificial intelligence: I. Themes and case studies of knowledge engineering. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977, 1014–1024.

11. Goldstein, I. and Papert, S., Artificial intelligence, language, and the study of knowledge. *Cognitive Science* **1** (January 1977) 54–123.
12. Green, C. C., *The application of theorem proving to question-answering systems* (Stanford University, Computer Science Department, AIM-96, August 1969).
13. Green, C. C., The design of the PSI program synthesis system. *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California, October 1976, 4–18.
14. Green, C. C. and Barstow, D. R., Some rules for the automatic synthesis of programs. *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisl, Georgia, USSR, September 1975, 232–239.
15. Green, C. C. and Barstow, D. 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, 1977) 335–359.
16. Green, C. C. and Barstow, D. R., On program synthesis knowledge, *Artificial Intelligence* **10** (November 1978) 241–279.
17. Kant, E., A knowledge based approach to using efficiency estimation in program synthesis. *Sixth International Joint Conference on Artificial Intelligence*, Tokyo, Japan (1979).
18. Knuth, D. E., *The Art of Computer Programming, Combinatorial Algorithms* (Vol. 4) (Addison-Wesley, 1977). (Preprint).
19. Lenat, D. B., The ubiquity of discovery, *Artificial Intelligence* **9** (December 1977) 257–286.
20. Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C., Abstraction mechanisms in CLU. *Commun. ACM*, **20** (August 1977) 564–576.
21. Low, J., *Automatic coding: choice of data structures* (Stanford University, Computer Science Department, AIM-242, August 1974).
22. Ludlow, J., Masters Project (Stanford University, 1977).
23. Manna, Z. and Waldinger, R., Synthesis: dreams ⇒ programs. (SRI International, Technical Note 156, November 1977).
24. Reiser, J. F. (Ed.) SAIL Reference Manual (Stanford University, Computer Science Department, AIM-289, August 1976).
25. Schwartz, J. T., On programming: an interim report on the SETL project (New York University, Courant Institute of Mathematical Sciences, Computer Science Department, June 1975).
26. Shortliffe, E. H., *MYCIN: Computer-Based Medical Consultations* (American Elsevier, New York, 1976).
27. Simon, H. A., Experiments with a heuristic compiler. *Journal of the ACM* **10** (April 1963) 493–506.
28. Sussman, G. J., *A Computer Model of Skill Acquisition* (American Elsevier, New York, 1975).
29. Teitelman, W., INTERLISP Reference Manual (Xerox Palo Alto Research Center, Palo Alto, California, December 1975).
30. Thorelli, L.-E., Marking algorithms. *Behandling Informations-tidskrift for Nodisk* **12** (1972) 555–568.
31. Waldinger, R. J. and Lee, R. C. T., A step toward automatic program writing. *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, D.C., 1969, 241–252.
32. Winston, P. H., Learning Structural Descriptions from Examples, in: Winston, P. H. (Ed.), *The Psychology of Computer Vision* (McGraw-Hill, 1975).
33. Wulf, W., London, R. and Shaw, M., An introduction to the construction and verification of ALPHARD programs. *IEEE Transactions on Software Engineering* (December 1976) 253–265.