

A KNOWLEDGE-BASED SYSTEM FOR AUTOMATIC PROGRAM CONSTRUCTION

David Barstow
Computer Science Department
Stanford University
Stanford, California 94305

ABSTRACT: PECOS is a system that synthesizes programs from abstract specifications through the use of a knowledge base of programming rules. The rules deal explicitly with various aspects of the programming process, including intermediate-level constructs and certain design decisions. Programs are specified as abstract algorithms in a high-level language for the domain of symbolic computation. Programs are constructed by a process of gradual refinement: PECOS retrieves and applies relevant rules until the original abstract description has been refined into a concrete program. When several rules are relevant in the same situation, each is applied separately, enabling PECOS to construct a variety of programs for a given abstract specification.

KEYWORDS: Automatic programming, Program refinement, Programming knowledge, Rules about programming, Knowledge-based systems.

1. INTRODUCTION

PECOS is a system that uses a knowledge base of programming rules to synthesize programs from descriptions of abstract algorithms [Barstow 1977a]. The rules embody knowledge about a variety of techniques for implementing data structures and algorithms. This allows PECOS to construct several distinct implementations of the same abstract algorithm. The implementation techniques dealt with by the rules include linked list and array representations of sets, simple table-oriented techniques for representing correspondences, and several types of enumeration algorithms with varying degrees of efficiency.

Programs are specified in a high-level language whose constructs are drawn from the area of symbolic programming. These include such information structures as sets and correspondences and such operations as testing for membership in a set and computing the inverse of a correspondence. Programs are synthesized by a process of gradual refinement. PECOS repeatedly retrieves and applies rules from its knowledge base until the original abstract description has been refined into a concrete implementation in the target language. Currently the target language is LISP (in particular, a subset of INTERLISP [Teitelman 1975]). When several rules are relevant in the same situation, PECOS applies each separately, resulting in the synthesis of a distinct program for each rule. PECOS can thus construct several implementations from one specification.

This ability is central to the role of PECOS as the Coding Expert of the PSI (Ψ) program synthesis system [Green 1976]. Loosely speaking, the Coding Expert's task is to enable the construction of a variety of implementations of the specifications produced by Ψ 's acquisition phase¹. The

¹ PECOS's specification language is that used by Ψ 's Program Model Builder [McCune 1977].

other module of the synthesis phase, an Efficiency Expert known as LIBRA, is used to select from among these alternatives.

The development of PECOS has required research along four directions: (1) the codification of a body of knowledge about programming; (2) the development of a model of program construction appropriate for that knowledge; (3) the design of a representation scheme; and (4) the design and implementation of a system that can apply the knowledge to produce implementations of abstract algorithms. No one aspect can be totally isolated from any of the others, but this paper will focus on the last. Overviews of the knowledge base and the model of program construction will be given, followed by a brief description of the representation scheme. Finally, PECOS's control structure will be discussed and an example of PECOS in operation will be given.

2. THE KNOWLEDGE BASE

Most of PECOS's abilities are based on its access to a large store of rules about programming. A sample of these rules is presented below. For clarity, the rules are given in English; a brief discussion of the internal representation will be given later.

RULE1: A collection may be refined into an explicit collection.

RULE2: A sequential collection may be refined into a linked list.

RULE3: If a linked list is represented as a LISP list, a test of whether an item is stored in an element cell of the list may be refined into a call to the LISP function MEMBER.

RULE4: If a collection is represented as a correspondence of items to boolean values, a test of whether an item is an element of the collection may be refined into a retrieval of the value corresponding to the item.

RULE5: If there is an ordering relation for the elements of a sequential collection, a test of whether an item is stored in some location in the collection may be refined into an enumeration of the elements in the collection, in which the enumeration order is based on the relation and in which the enumeration may be terminated early if an item is found which follows the target item according to the relation.

RULE6: If the enumeration order is the same as the stored order of a collection, the state of the enumeration may be saved as a position in the collection.

RULE7: If the enumeration order is ordered by an ordering relation and the stored order in a sequential collection is ordered by the same relation, the enumeration order is the same as the stored order.

RULE8: If there is an ordering relation for the elements of a sequential collection, the elements may be stored according to that relation.

RULE9: An ordering relation for integers is "less than."

RULE10: A scheme for remembering a value computed in one place and used elsewhere is to bind the value to a variable.

RULE11: If a value is remembered as the value of a variable whose name is X, the value may be accessed by retrieving the value of X.

Several observations about the rules are in order. First, note that the rules mention various programming concepts explicitly. The most abstract concepts are those included in PECOS's specification language (e.g., "collection", "correspondence", "is-element"). The most concrete are those used in the target language (e.g., "LISP list", "MEMBER function"). Some concepts (e.g., "linked list", "position in a sequential collection") represent intermediate notions, more concrete than the specification language and more abstract than the target language. Other concepts (e.g., "enumeration order", "scheme for remembering a value") represent design decisions involved in symbolic programming. Among the benefits of such abstraction levels are significant amounts of knowledge-sharing (e.g., knowledge about the common aspects of linked lists and arrays is dealt with at the level of sequential collections) and relatively simple extensibility (different abstractions provide a variety of hooks for new rules).

Second, note that refinement plays a central role in the rules. RULE2, for example, describes "linked list" as a refinement of "sequential collection." The essence of such a relationship is that the refined concept (linked list) is a more concrete notion than the abstract concept (sequential collection). PECOS uses such rules to drive the refinement process which produces the final program: the application of RULE2 to a given sequential collection represents a decision to implement it as a linked list. The validity of such an implementation is often dependent on other parts of the program. For example, RULE4 specifies that a "get-correspondent" operation is a refinement of an "is-element" operation only if the collection has been refined into a correspondence.

Third, note that the rules are all relatively small. One major motivation for designing the rules this way is that individual rules are intended to reflect individual decisions and choices, rather than groups of choices, thereby making it easier to focus on the nature of each decision. A second motivation is that the rules are then relevant in wide ranges of situations.

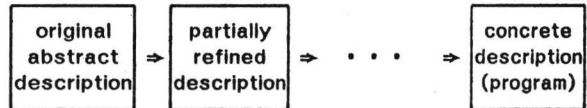
Finally, note that some of the rules deal with programming in general and others are specific to programming in LISP. For example, RULE2 could be considered a general rule while RULE3 could be considered a LISP rule. One of the benefits of such a separation is that the replacement of the LISP rules by rules for some other language might enable PECOS to write programs in that language. A set of rules for SAIL (an ALGOL-like language) is currently being developed. A few simple SAIL programs have been written using a combination of previously developed general rules and the newly developed SAIL rules [Ludlow 1977].

Considerable effort has gone into the codification of this knowledge base in the hope of developing a set of rules reflecting notions and decisions inherent in the programming process (although often in the guise of hidden assumptions). Taken together, the rules may be considered to be a step toward a "science" of programming: they are a coherent collection of facts, specifying certain concepts and relationships between them, which describe part of the process of programming within the domain of symbolic computation. The explication of such knowledge has been a major goal of this line of research [Green and Barstow

1976, 1975]. The PECOS system demonstrates that such a body of knowledge can provide the basis of an automatic programming system; in fact, the rules were developed with this application in mind. However, it is hoped that the concepts and relationships are sufficiently relevant to other aspects of programming that other applications may be found. An interesting possibility might be to investigate their usefulness in the teaching of programming. For the interested reader, a description of the entire collection of rules may be found elsewhere [Barstow 1977a].

3. A REFINEMENT MODEL OF PROGRAM SYNTHESIS

PECOS constructs programs through the application of rules from its knowledge base. This process is based on a model of program synthesis as gradual refinement. This 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 implementation in the target language. Such sequences will be referred to as *refinement sequences* and the individual descriptions will be referred to as *program descriptions*. The process of deriving one description from the previous one will be referred to as a *refinement step*.

We may illustrate this through a very simple example (more typical cases often require hundreds of refinement steps). Suppose the original specification is:

Is X (integer) an element
of Y (collection of integers)?

(In order to focus on the refinement process, we will use English phrases for program descriptions. The internal form of these descriptions is a collection of nodes and properties as will be discussed in more detail in the next section.) The first refinement step might be to decide to implement the collection (Y) as an explicit collection (as opposed to some form of implicit representation, such as upper and lower bounds):

Is X (integer) an element
of Y (explicit collection of integers)?

The next step might refine the "is-element" operation. An appropriate refinement for explicit sets would be:

Is X (integer) explicitly an element
of Y (explicit collection of integers)?

The next step might be to represent the explicit collection as a stored collection (as opposed to markings on the elements, for example):

Is X (integer) explicitly an element
of Y (stored collection of integers)?

with a corresponding refinement for the operation:

Is X (integer) stored as an element
of Y (stored collection of integers)?

The next decision might be to implement the stored collection as a sequential collection (in contrast with a tree representation):

Is X (integer) stored as an element of Y (sequential collection of integers)?

Again, there is a parallel refinement for the operation:

Is X (integer) stored in a location of Y (sequential collection of integers)?

Next, we might decide to implement the sequential collection as a linked list (rather than, say, an array):

Is X (integer) stored in a location of Y (linked list of integers)?

with the corresponding refinement of the operation:

Is X (integer) stored in an element cell of Y (linked list of integers)?

Note that up to this point, no use has been made of the fact that the target language is LISP. For example, the linked list could still be represented using a parallel array representation, one holding the items and one holding the links. However, in the interest of simplicity, we will let the next refinement step be the decision to use a LISP list implementation for the linked list:

Is X (integer) stored in an element cell of Y (LISP list of integers)?

At this point, the refinement of the operation is dependent on the refinements of X and the elements of Y. We will suppose that both are implemented as LISP integers, producing the next two refinement steps:

Is X (integer) stored in an element cell of Y (LISP list of LISP integers)?

Is X (LISP integer) stored in an element cell of Y (LISP list of LISP integers)?

Finally, the operation can be refined into a call to the LISP function MEMBER:

(MEMBER X Y)

We thus have a sequence of twelve refinements taking the original specification and producing a LISP expression that implements it. Note that the refinements at each step involve only very small changes. This is characteristic of the programming rules upon which PECOS is based. A deliberate effort has been made to identify and separate out decisions that are often implicit. Thus, the complete refinement of the collection Y took five steps, each corresponding to a particular design decision. Note also that some steps may require other steps to have been made previously. For example, the refinements which led from the "is-element" operation to a call to MEMBER each required the data structure to have been refined in a particular way. Although it is quite common for the refinements of operations and data structures to proceed in

such a parallel fashion, it need not always be the case. One part of a program may be refined to great detail before another part is considered. In addition, several algorithms might be appropriate for the same data structure, just as the same algorithm may be applicable to several representations. For example, a linear scan with an explicit loop would have served just as well as the call to MEMBER.

Some of the rules shown in the previous section could be used to derive parts of the refinement chain we have just seen. For example, RULE1 could be used to refine the first diagram into the second, RULE3 could be used to produce the final refinement step, and so on.

4. INTERNAL REPRESENTATIONS

Before considering the details of how PECOS uses its knowledge base to produce refinement sequences, a few details of the internal representation must be presented.

4.1. Descriptions in a Refinement Sequence

Internally, PECOS represents each description as a collection of nodes. Each node is labelled with its CONCEPT. (As mentioned earlier, these concepts identify particular programming notions.) Thus, a node labelled IS-ELEMENT represents an operation testing whether some item is in a particular collection. In addition to the label, each of the nodes has a set of properties related to the node's concept. For example, an IS-ELEMENT node would have properties for each of its arguments. (The properties are named ELEMENT and COLLECTION.) Although property values may be arbitrary expressions, they are frequently links to other nodes (as in the IS-ELEMENT case). Figure 1 shows part of the internal representation for the expression IS-ELEMENT(X, INVERSE(Y, Z))².

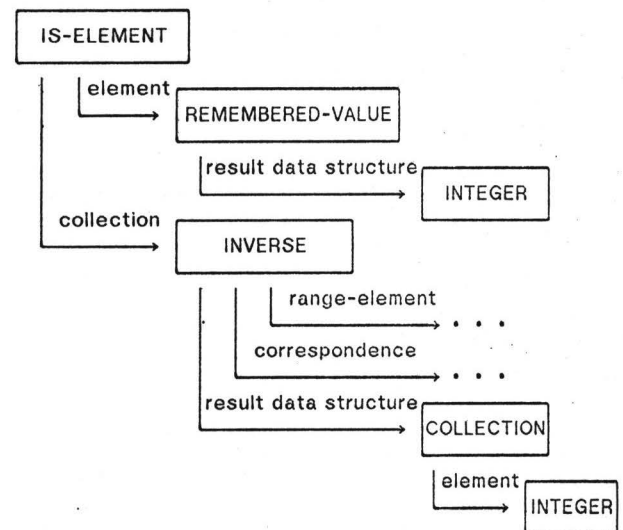


Figure 1

Each box is a node. The labels inside the boxes are the concepts and the labeled arrows are properties. In the refinement sequence presented earlier, the arguments to the IS-ELEMENT operation were given merely as X and Y. In general, such arguments are links to nodes representing operations which return values to be used as the arguments. (The values are indicated by the RESULT-DATA-STRUCTURE property.) In this case, one of

² Read "Is X an element of the inverse mapping of Y under the correspondence Z?".

these operations is a REMEMBERED-VALUE node, specifying that the value to be used is computed elsewhere in the program. The other operation is an INVERSE node. Note that the result of the inverse operation is represented explicitly by the COLLECTION node, although it is only implicit in the corresponding English description. This node represents the data structure passed from the inverse operation to the IS-element operation. Rules for refining INVERSE generally have conditions on the data structure produced, to insure that the representation is what the refinement produces. Likewise, rules for refining IS-ELEMENT generally have conditions on the argument data structure in order to insure that the refinement can accept that representation for its arguments. Since the same node (the COLLECTION node) is used in both cases, the two refinements are coordinated.

There are only a few types of changes that may be made in a single refinement step: the introduction of new nodes, the addition of properties to existing nodes, and the refinement of one node into another³. Other types of changes (e.g., modification of an existing property) may be imagined, but no use for them has yet been found.

4.2. Rules in the Knowledge Base

The internal representation of the programming rules is based on two observations. First, the changes made in each refinement step are generally quite small and are of certain specific types. Second, the 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 instances of the types of modifications just mentioned. These observations lead to a classification of rules into a set of rule 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. RULE1, RULE2, RULE3, and RULE5 are refinement rules.

Property rules attach a new property to an already existing node. RULE9 and RULE10 are property rules. Property rules are often used to indicate explicit decisions which guide the refinements of distinct but conceptually linked nodes. For example, RULE10 specifies how references to a data structure are to be refined.

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. RULE7 is a query rule.

The internal representation of the rules is based on this categorization of rule types:

(REF+ <node pattern> <refinement node>)

(PROP+ <property name> <node pattern> <value>)

(QUERY+ <query pattern> <query answer>)

where REF+, PROP+, and QUERY+ are tags indicating rule type. In REF+ and PROP+ rules, each <node pattern> consists of a <concept> and an <applicability pattern>.

³ Although it is not strictly accurate, "refinement" may be considered to mean the replacement of the more abstract node by the refined node. For various reasons, PECOS actually retains the abstract node as well as the refinement node.

Several other issues have arisen in the design of the rule base organization, including an indexing scheme for efficient rule retrieval, the details of the pattern matcher, and the breakdown of the condition patterns into separate parts for different uses. Detailed discussions of these issues may be found elsewhere [Barstow 1977a, 1977b].

It should 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 [Manna and Waldinger 1977]) 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. CONTROL STRUCTURE

PECOS uses a relatively simple task-directed control structure to develop refinement sequences for a given specification: in each cycle, a task is selected and a rule applied to the task. When several rules are applicable, a refinement sequence can be split, with each rule applied in a different branch. PECOS thus actually 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 sequence⁴. 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 (corresponding to the types of changes involved in refinement steps and to the different rule types):

(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 (REF+ (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.

With its current knowledge base, PECOS usually succeeds at achieving each task. (Hence, most refinement sequences lead to complete programs.) The major reason for this lies in the knowledge base: there are refinement rules for each intermediate level concept, and property rules for each of the necessary properties. However, some sequences lead to situations where no rules are applicable, and PECOS abandons such lines of development.

In the interests of brevity, further details of the control structure and context mechanism used for the tree of descriptions will not be considered here. More detailed discussions may be found elsewhere [Barstow 1977a, 1977b].

⁴ The nature of such refinement trees is central to the practicality of this approach and will be discussed in section 7.

6. PECOS IN OPERATION

Let us now look at PECOS implementing a different algorithm for the IS-ELEMENT operation, in order to illustrate the nature of the tasks and the way that PECOS focuses design decisions. Recall that the original specification was:

Is X (integer) an element
of Y (collection of integers)?

We will pick up the refinement process at the point where Y has been refined into a SEQUENTIAL-COLLECTION, and where the IS-ELEMENT operation has been refined into an IS-STORED-IN-A-LOCATION operation. The node and property representation is given in Figure 2 (with several nodes omitted in order to emphasize the important ones). The numbers on the left will be used to refer to the nodes.

The task is now (REFINE 1), that is, to refine the IS-STORED-IN-A-LOCATION node. In the example in section 3, we followed the path where this was refined into a test on whether X was stored in an element cell of Y. Another refinement rule, RULE5, is applicable here, but only if there is an ordering relation for the elements of the sequential collection Y, so a subtask is established: (PROPERTY ORDERING-RELATION 4). One rule for this task, RULE9, is applicable only if the items are integers, as is the case here. PECOS applies this rule and LESS-THAN is attached as the ORDERING-RELATION property of node 4. Having achieved the subtask, PECOS reconsiders the original task (REFINE 1), and RULE5 is now applicable. This produces an ENUMERATE-ITEMS node, where the enumeration is constrained to be in the order specified by the ordering relation associated with the elements of Y (LESS-THAN) and where the search may be terminated prematurely (before all the elements of Y have been examined) if an element is found which follows X under the same ordering relation. Constraints such as these are specified by attaching properties to the ENUMERATE-ITEMS node, as illustrated in Figure 3.

In turn, the ENUMERATE-ITEMS node (5) is refined into a loop structure which uses an ENUMERATION-STATE node to determine how far the enumeration has proceeded. The refinement of this node coordinates the refinements of the parts of the program that refer to it (initialization, termination test, incrementation), as illustrated in Figure 4. Note how nodes 7, 8, and 9 all have links to node 6. This is typical of the way that PECOS focuses design decisions: the refinements of the various parts are all dependent on a single node.

When PECOS tries to refine the ENUMERATION-STATE node, RULE6 is applicable only if the elements are to be enumerated in the order in which they are stored. This causes a subtask to be generated: (QUERY STORED-ENUMERATION-ORDER 6). The ENUMERATION-ORDER has already been constrained to be ORDERED by the LESS-THAN relationship. All that is needed, then, is to determine whether the sequential collection is also ordered. The subtask for this is: (PROPERTY STORED-ORDER 3). Two rules are applicable here, and we will follow RULE8, which specifies that the items may be ordered if there is an ordering relation for them. Since an ordering relation, LESS-THAN, has already been found, RULE8 can be applied, resulting in (ORDERED LESS-THAN) as the STORED-ORDER property of node 3, a

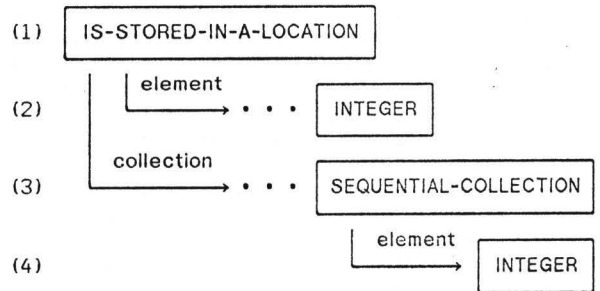


Figure 2

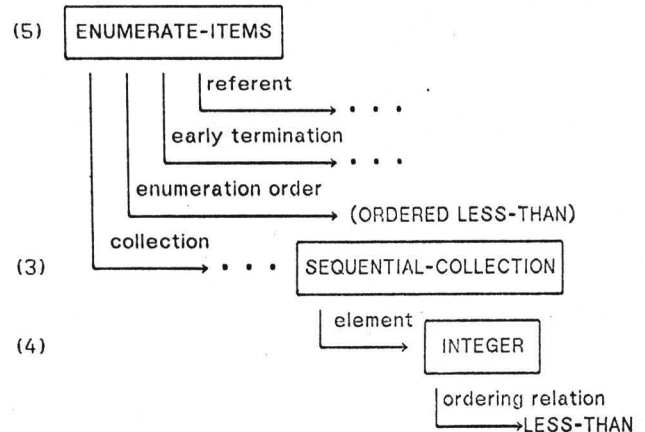


Figure 3

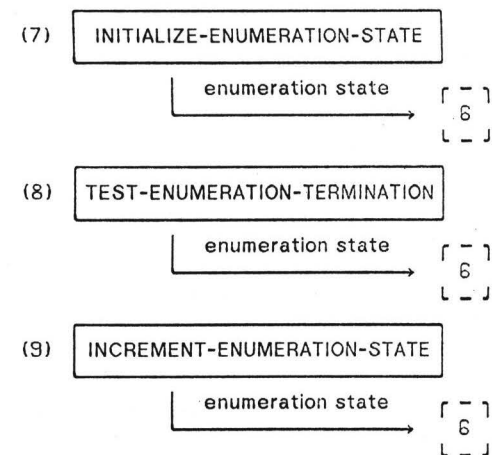
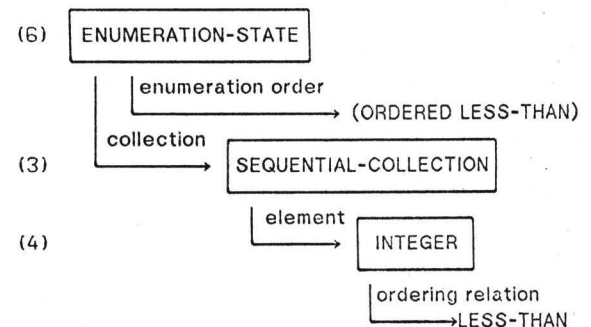


Figure 4

further specification on the representation of Y^5 . Now RULE7 can provide the answer, TRUE, to the query, and RULE6 produces a POSITION-IN-COLLECTION node as a refinement of node 6. The rest of the refinement path is relatively straightforward. Note, however, that at this point Y still has not been refined further than a sequential collection. All of the rules that have been used are independent of the particular implementation chosen for the sequential collection; linked lists and arrays would work equally well. This is an example of the way the rules in the knowledge base have been explicitly designed to operate at the highest possible level of abstraction. Before the program can be completed, of course, this decision must be made, and we will assume that, again, a linked list is chosen. This is the last interesting aspect of the refinement sequence. For the curious, the final program looks like the following (with mnemonic labels for increased readability):

```
(PROG (REST ITEM)
      (SETQ REST Y)
      RPT (SETQ ITEM (CAR REST))
      (COND
        ((OR (NULL REST)
              (ILESSP X (CAR ITEM))))
        (RETURN NIL))
        ((EQP (CAR ITEM) X)
         (RETURN T)))
      (SETQ REST (CDR REST))
      (GO RPT))
```

If the IS-ELEMENT operation were embedded in the specification of a larger program, the decision to represent Y as an ordered sequential collection would presumably have wider ramifications, possibly including the necessity of sorting Y. This is characteristic of the programs that PECOS writes: multiple representations are often considered for what in the original specification was the same data structure, with the program converting representations as necessary.

7. A SPACE OF PARTIALLY IMPLEMENTED PROGRAMS

One of the motivations for this work is the observation that different implementations of abstract concepts are appropriate in different situations. PECOS's refinement tree represents a space that can be searched for such an "appropriate" implementation. In fact, the ability to construct such a space was one of the primary goals in developing PECOS's rules⁶. Since determining the appropriate implementation involves exploring this space, the size of PECOS's refinement trees is a critical issue.

⁵ Note that when the decision is taken the other way (to leave the elements unordered), the refinement path requires the construction of a considerably more complicated enumeration, one which searches the collection at each stage for the smallest element, then compares that to X, and so on. The result is an implementation of the IS-ELEMENT operation that requires on the order of n^2 comparisons!

⁶ Note that the nodes of this space all represent semantically correct programs; that is, they all implement the original specification, although at varying levels of detail. The use of refinement rules enables PECOS to proceed directly toward correct implementations, without any need for testing different combinations of primitive constructs to see if they are correct.

One aspect of the size is the length of the individual refinement sequences (i.e., the number of rule applications). This varies greatly, but preliminary experiments indicate that about two rule applications are required per CONS cell in the S-expression for the final program. Thus, if the correct choice can be made without expanding any alternative paths, the cost of constructing the best implementation is roughly linear with the size of the final program. This cost may be reduced through the use of special purpose rules to handle frequently occurring cases. Such rules can collapse an entire sequence of rule applications into a single rule. While the rules to derive the sequence must still be available for use in other situations, one can frequently get by with a "short-cut" rule. PECOS currently has a few such rules and more will be added soon.

In an effort to reduce the entire space, PECOS currently postpones consideration of all choicepoints for as long as possible⁷. This technique, together with the use of several levels of abstraction, results in trees that tend to be "skinny" at the top and "bushy" at the bottom. In the membership test, for example, by delaying the refinement of the sequential collection (into either a linked list or an array), 126 refinement steps were made before working on the choice; following the split, 16 further steps were required in the linked list case and 51 steps in the array case. Had the split been made when the task was first considered, splitting would have occurred after 9 steps, leaving 133 left in the linked list case and 168 in the array case. Thus, delaying the choicepoint cut the total number of rule applications in half.

The techniques mentioned above are intended to reduce the total size of the space. Another simple technique helps to reduce the amount of the space that must be searched. When two parts of the program are totally independent, and both involve choices, the entire space represents the cross-product of the alternatives in the two cases. However, under the assumption that the independence of the parts enables a choice for one to be made independently of any work that has been done on the other, the exploration of some of the paths can be avoided. The basic idea is that one of the choicepoints be carried through to conclusion and one of its alternatives be chosen before considering the other choicepoint. Then the alternatives for the second choicepoint need only be considered in the chosen path for the first.

Although these simple techniques can help to reduce the size of the space, the problem of actually making choices still remains. Perhaps the greatest benefit of the use of abstraction and refinement is the way that such choice-making is facilitated. Since each step is quite small, the effects of alternative branches can be relatively easily understood. It is usually unnecessary to complete all implementations in order to determine which alternative is preferable. In addition, the uniform representation of descriptions in refinement sequences facilitates the use of analytic planning techniques based on cost estimators for the unrefined parts. Work on LIBRA (Ψ 's Efficiency Expert) has included the development of such heuristic and analytic techniques for making choices [Kant 1977]. PECOS and LIBRA have been interfaced and now operate as Ψ 's synthesis phase. A discussion of their interaction, as well as a detailed example of their joint operation, is available elsewhere [Barstow and Kant 1976].

⁷ Several other artificial intelligence programs have employed similar tactics (e.g., NOAH [Sacerdoti 1975]).

8. DISCUSSION

The use of large amounts of detailed knowledge has become an increasingly important paradigm for developing artificial intelligence systems. However, most approaches to automatic programming have concentrated on the development of relatively general techniques (e.g., SYNSYS [Manna and Waldinger 1977]). PECOS's rules enable the construction of larger and more varied programs, but the price paid is a loss of generality. The "ultimate" automatic programming system will presumably employ both techniques. Although current trends in programming methodology emphasize abstraction and refinement, most automatic programming work has not used intermediate level concepts extensively. Low's data structure selection system involves mapping directly from abstract concepts to machine-level constructs [Low 1974]. Most problem-solvers deal directly with the primitive operations of the target language.

Currently (May 1977), there are about two hundred rules in PECOS's knowledge base. In the next few months, this number will increase to about three hundred. This enlargement of the rule base will concentrate on higher level knowledge (e.g., alternative set representations), as the current rules involving low level knowledge should be able to handle most of the low level details. Even with its current rule base, PECOS is able to construct a variety of implementations for specifications of many different programs. In theory, PECOS could construct implementations for any program expressible in the high-level specification language. In practice, time and space limitations prevent the consideration of programs whose specification is more than about "a page." The number of implementations possible for a given program specification varies considerably with many factors (for instance, whether a particular primitive is known to be numeric or symbolic), so the best indication is perhaps to take a simple example. PECOS can implement the "is-element" program considered in this paper in about a dozen ways, with the principle variations being the use of linked lists, arrays, or very simple hash tables for the data structures, and the use of several types of enumeration for the algorithms. The most complex program that PECOS has implemented is one involving manipulation of an inverted correspondence (i.e., indexed by range values). The specification was about a half page in length and the implementation was about 100 lines of INTERLISP code.

The development of PECOS represents the final stage in an experiment investigating a knowledge-based approach to automatic program construction. The essence of this approach involves the identification of concepts and decisions involved in the programming process and their codification into individual rules. These rules are then represented in a form suitable for use by an automatic programming system. The benefits of this approach seem to lie in the variety of programs (with respect to both data structures and algorithms) that can be implemented. A side benefit is that the rules themselves constitute a detailed explication of knowledge about symbolic programming. PECOS demonstrates that such a knowledge base can serve as the basis of an automatic programming system capable of constructing a variety of implementations for abstract specifications.

9. ACKNOWLEDGEMENTS

Cordell Green, my thesis advisor, 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 and Jorge Phillips have provided very helpful comments on earlier drafts of this paper.

10. REFERENCES

- [Barstow 1977a]
Barstow, David R. *Automatic construction of algorithms and data structures using a knowledge base of programming rules*. forthcoming Ph.D. thesis, Stanford University, 1977.
- [Barstow 1977b]
Barstow, David R. *A knowledge base organization for rules about programming*. Workshop on Pattern-directed Inference Systems, May 1977 (to appear in SIGART, June 1977).
- [Barstow and Kant 1976]
Barstow, David R., and Kant, Elaine. *Observations on the interaction of coding and efficiency knowledge in the PSI program synthesis system*. Proceedings of the Second International Conference on Software Engineering, San Francisco, California, October 1976, pages 19-31.
- [Green 1976]
Green, Cordell. *The design of the PSI program synthesis system*. Proceedings of the Second International Conference on Software Engineering, San Francisco, California, October 1976, pages 4-18.
- [Green and Barstow 1976]
Green, C. Cordell, and Barstow, David R. *A hypothetical dialogue exhibiting a knowledge base for a program understanding system*. In Elcock, E. W., and Michie, D. (Eds.). *Machine Representations of Knowledge*. Ellis Horwood Ltd. and John Wiley, 1976.
- [Green and Barstow 1975]
Green, C. Cordell, and Barstow, David R. *Some rules for the automatic synthesis of programs*. Advance Papers of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi, Georgia, USSR, September 1975, pages 232-239.
- [Kant 1977]
Kant, Elaine. *The selection of efficient implementations for a high level language*. Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, August 1977.
- [Low 1974]
Low, J. *Automatic coding: choice of data structures*. Stanford University, Computer Science Department, AIM-242, August 1974.
- [Ludlow 1977]
Ludlow, Juan. Masters Project. Stanford University, 1977.
- [Manna and Waldinger 1977]
Manna, Zohar and Waldinger, Richard. *The automatic synthesis of recursive programs*. Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, August 1977.
- [McCune 1977]
McCune, Brian P. *The PSI program model builder: synthesis of very high-level programs*. Proceedings of ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages, August 1977.
- [Sacerdoti 1975]
Sacerdoti Earl. *The nonlinear nature of plans*. Advance Papers of the Fourth International Joint Conference on Artificial Intelligence. Tbilisi, Georgia, USSR, September 1975, pages 206-214.
- [Teitelman 1975]
Teitelman, Warren. *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, California, December 1975.