DATA STRUCTURES THROUGH PLAN INSTANTIATION

Lee A. Becker
Department of Computer Science
Tulane University
New Orleans, Louisiana 70118

INTRODUCTION

One of the skills that we want students in our Data Structures courses to acquire is the ability to take an implementation data structure and an operation or task to be performed on it and to produce an algorithm; this skill is illustrated in Figure 1.

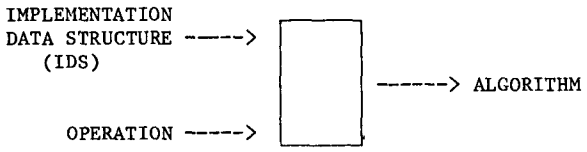


Figure 1

Often this is embedded in a more complex task, as illustrated in Figure 2.

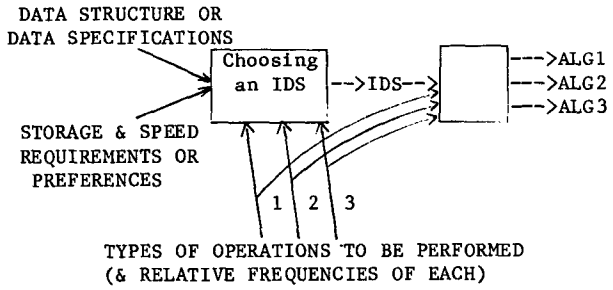


Figure 2

Of course, there may be many successive implementations or mappings of one data structure into another. In practice, however, at the level of a Data Structures course, one or two such mappings is all that is required. Essentially, the sequence of mappings is bounded at both ends. Students in such a course need not consider, or can take as given, the mapping(s) from constructs available in an algorithm design language or high level programming language down to the level of the machine. At the other end, students, predictably, have not yet been exposed to data structures more abstract than those that are presented explicitly in the course.

The skill in Figure 1. provides the focus of this paper. To confirm to oneself the importance of this skill, consider how often examination contain questions of the form: given a data structure X, write an algorithm that will Y it.

Consider now the learning task confronting the student. The student is presented with a stream of triples (DATA STRUCTURE,OPERATION,ALGORITHM) and is asked to acquire a machine (function, black box, skill) that will take a given data structure and operation and produce an algorithm. In essence, the student is normally presented with examples of inputs and corresponding outputs of a machine like the one that he/she is to acquire. The goal of this paper is to demonstrate the that **it would be desirable for students to acquire the kind of knowledge that would allow them to accomplish the task represented in Figure 1 by ˝plan instantiation˝.** To use plan instantiation the machine would have to contain a set of ˝plans˝, one for each type of operation, and include some capability for ˝instantiating˝ plans. A plan can be thought of as a sequence of steps to accomplish a goal. Instantiating can be thought of as applying a plan to a given situation.

WHY NOT ALTERNATIVE, WEAKER METHODS OF PROBLEM SOLVING?

The capability for instantiating plans is presumably used in many domains, including everyday life. It forms a basis for several recent theories of problem solving (McDermott[4] and Wilensky[8]). Within a theory that included a variety of problem solving strategies, it would be reasonable to assume that plan instantiation would be the most frequently used strategy; presumably the reason for this is that plan instantiation is the method which involves the least cognitive effort. Before discussing the merits of plan instantiation, we must consider the alternatives.

It is clear that plan instantiation cannot be the sole method or technique for problem solving, since plan instantiation could only be applicable when one possessed a plan which could be used for

the task at hand. The other methods would include analogical reasoning, means-ends analysis, and using general plans (cf. Carbonell[1]). In analogical reasoning a known plan for a similar problem is transformed so that it might be applicable to the problem at hand. Means-ends analysis is a general problem solving technique, which given a goal and a set of possible plan steps (primitive operators) with the preconditions and effects of each, searches for a sequence of steps to accomplish a goal by attempting at each point to take the step which maximally decreases the 'distance' to the goal. General plans can be used for decomposing a complex task into subtasks, upon which any of the methods can be subsequently used. The implied division between 'general' plans, which primarily break up a problem into subproblems, and 'specific' plans, which can merely be instantiated, in actuality is more of a dimension. We will define below the characteristics of the type of plan we advocate.

It is useful to point out that **even** 'weaker' techniques than means-ends analysis are conceivable, for example generating in turn all sequences of steps and testing to see if they would solve the problem. Each of these search techniques, including means-ends analysis, involves trial and error. In analogical reasoning, the appropriate transformations are not known ahead of time, and thus this is also a matter of trial and error; in fact, it has been even explicitly formulated as a means-ends search problem, where the possible steps are plan transformations (Carbonell[2]). It is also clear that considerable effort and time can be expended on such trial and error methods.

For means-ends analysis, at least, essentially no use is made of previous problem solutions. There is an attendant time-space tradeoff here: while computation is considerable, storage for previous problem solutions is minimized. It is not clear, however, that long term memory presents any storage limitations (the limited capacity of short term storage does have significant repercussions; see below). Thus we will consider the other end of the spectrum - complete memorization.

WHY NOT COMPLETE MEMORIZATION?

It is clear that we do not think our students should just memorize triples. This can be evidenced by the advice, caution or warning we are always giving to our students - not to memorize, but to try to 'understand' the algorithms. We know, of course, that memorization of triples cannot provide the students with capabilities to write algorithms for unfamiliar data structures or not previously encountered pairings of familiar data structures and operations.

We assume that one reason that simple, complete and linear memorization is not employed is a size limitation for a 'psychological module'. We suggest that the number of steps in any plan must be quite small, and that it is bounded by the size of short term memory, roughly 7 + or - 2. One situation in which the limitation of short term memory size would manifest itself is during acquisition. The absolute maximum size of this psychological module can be contrasted with that of a module in a program which is generally taken to be one page or two pages. In the latter case the absolute size limitation is motivated by the consideration of readability; in particular, it allows the module to be examined in the program document without turning pages. In a sense the motivation for the size limitation of the psychological module is the identical, only the 'window' is smaller. It should be noted that a small module size is particularly conducive to top-down design, and it has been my experience that students (at least at the lower levels) do a better job of problem solving (designing), if the size of their modules is very strongly restricted, for example to a maximum of four steps.

Psychological modules or plans must be logically coherent, and the most important criterion for grouping sequences of steps into a plan is that they occur together often. These occurrences are either in plans we construct and perform or in sequences of actions that we observe or, in the case at hand, in algorithms to which we are exposed. As a by-product of grouping into these psychological modules we reduce total storage requirements, since the sequence of steps in the module is specified only a single time. Second, and more importantly, in this manner we acquire the ability to encode plans and accomplish ever more complex tasks while remaining within the constraints imposed by the limited size of short term memory. Thirdly, these plans can be useful and applicable in dealing with situations we have not previously encountered, limiting very significantly the degree to which weaker problem solving techniques must be employed.

THE INSTANTIATION OF PLANS

Having motivated the extensive use of plans, let us now consider in some detail how plans are instantiated and look more closely at the nature of the plans themselves. In order to use plan instantiation, the acquired machine must contain a set of plans. For the case at hand, which plan to use would be determined by the operation to be accomplished. As pointed out above, as a by-product of grouping steps into plans, the amount of long term memory used is reduced. Earlier we suggested that there was no lack of storage space in long term memory, but by using generalized plans the total number of plans is reduced, and thus the task of identifying and accessing the plan to instantiate is reduced.

As noted above, the various instantiations of a single plan are not necessarily identical. Some of the steps are always instantiated identically, while the instantiation of other steps differs depending on the input parameter (here the data structure). What unifies each of the latter type of step is its function or goal. It is also only the latter type of step which requires any real cognitive effort to instantiate. In some cases this effort will be minimal, in others it will require a moment's reflection, while for others it might even involve a somewhat greater effort. In the case at hand, where the input parameter is a data structure, a drawing or schematic

representation of the data structure may be useful; in some cases, both before and after the operation, and each change could be thought of as a step.

To understand more fully the process of instantiation of plans, especially in the context of producing algorithms for operations on data structures, one should consider the 'plans' in the appendix. For presentation, the plans must be represented in some formalism, and we have chosen to use pseudo-code (generic procedures and functions). Recall that in the context with which we are concerned the output of the instantiation process will be an algorithm. The sections of the plans which are in bold-faced will never be part of the output algorithms. These sections may be thought of as embodying directives to the instantiator; they all refer to properties of the input data structure (prefixed with **Is-**). The underlined, non-bold-faced sections will be part of the output algorithms for some instantiations, while the non-bold-faced, non-underlined sections will appear in all output algorithms. In other words, the instantiator takes a plan and a data structure and outputs all of the non-bold-faced, non-underlined sections and (possibly) some of the non-bold-faced, underlined sections.

A number of assumptions have been made to simplify these 'plans'; for example, array indices have been assumed to have a range 1..NUMOFENTRY, doubly linked list have been assumed to be circular and traversed via RLINKs, and we have not included the capability for deleting from a binary search tree (BST). For clarity of presentation parameter passing has been ignored, and one should bear in mind that these are psychological modules and not subprogram units.

PLAN INSTANTIATION: THE HAPPY MEDIUM

In a real sense the instantiation of generalized plans stands between the simple complete memorization of each individual sequence of steps and the employment of the weaker methods. It lies closer to the former, and the use of the latter is localized and limited. At this point we can specify where along the general-specific dimension of plans the type of plan which is advocate here lies:

(1) Its instantiations differ only on the basis of an input parameter (upon which the plan operates); the input parameter here will be the data structure.

(2) There can be a sequence of top-down divisions into smaller tasks (plans into subplans), but only in the terminal plans, i.e. those which are not further subdivided (cf. the hierarchy chart), can there be any necessity for the weaker problem solving methods.

This paper is not the first to suggest the relevance of 'plans' for Computer Science Education; Soloway & Woolf[7], Mayer[3], and Peterson[6] all recognize the value of plan-based approaches to teaching. In this context one of the Perlis[5](#10, p.7) epigrams can be interpreted as advocating the use of plans:

> "Get into a rut early: Do the same processes the same way. Accumulate idioms. Standardize. The only difference(!) between Shakespeare and you was the size of his idiom list - not the size of his vocabulary."

The vocabulary here corresponds to the primitive possible steps. The idioms are the plans, i.e. the psychological modules. Shakespeare was successful (and prolific) because of the extent to which he was able to use plan instantiation (as opposed to, say, means-ends analysis).

This paper argues it is desirable and feasible that students in a Data Structures course acquire the ability to carry out the task represented in Figure 1 by plan instantiation. The reason that this approach is so well-suited to this task is because a data structure is a very appropriate input parameter to an instantiation process. We do not wish to advocate the use of any particular plans, but the sample plans are provided in the appendix to illustrate the generality and power of the approach. We also wish to emphasize that we do not advocate the acquisition of such plans done to the last detail. It is reasonable to use weaker methods, like before and after drawings, for some terminal subplans, for example NORMALINSERT. The crucial importance of acquiring higher level plans, like TRAVtoFINDWHERE, must be emphasized. In fact, TRAVtoFINDWHERE itself can be thought of as an instantiation of a high level plan TRAVto(DOIT):

```
GETFIRST
if NOT EMPTY then
  repeat
    [DOIT
     if NOT EXIT then ADVANCE]
  until EXIT or NOMORE
```

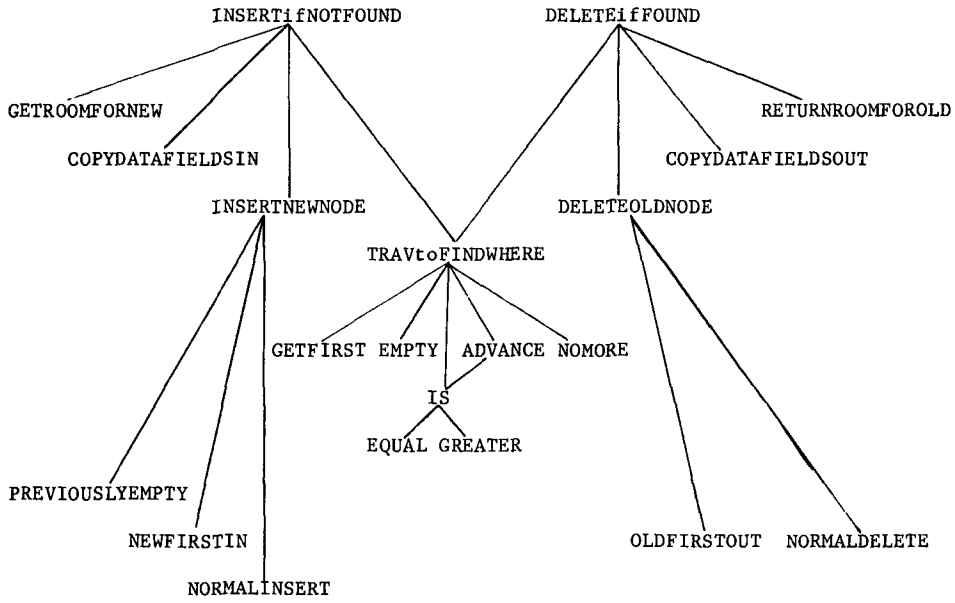
Here the procedure DOIT can set EXIT. Thus TRAVto(DOIT) is like a LISP MAP- function or Backus' functional form (apply to all), except that it also provides the possibility of exiting, for example in a case where DOIT involves searching.

With respect to the question of how to support or help the students acquire the ability to do the task represented in Figure 1 by plan instantiation, several suggestions can be made. In some cases plans should be presented explicitly. Such generic procedures and functions can often clearly illustrate important points; for example, they illustrate the value of headers in terms of algorithm simplicity. In other cases the students should be forced to extract, i.e. generalize the plans themselves. In this case the instructor must be consistent in the algorithms he/she presents; the instructor should formulate a plan and follow it in each algorithm. In particular, optimizations should be presented as modifications of a 'basic' algorithm, i.e. one derived from the general plan.

```
* * * procedure INSERTifNOTFOUND * * *
TRAVtoFINDWHERE
if NOT FOUND then
              [GETROOMFORNEW
               COPYDATAFIELDSIN
               if NOT IsARRAY then INSERTNEWNODE]

* * * procedure TRAVtoFINDWHERE * * *
FOUND <- FALSE
GETFIRST
if NOT EMPTY then
  repeat
    [if IS(INKEY,EQUAL) then
                          [FOUND <- TRUE
                           exit]
     if IsORDERED then if IS(INKEY,GREATER) then exit
     ADVANCE]
  until NOMORE

* * * procedure GETFIRST * * *
if IsARRAY then COUNTER <- 1
           else [if NOT IsDBLYLL then TRL <- INPTR
                 if HasHEADER then if IsSINGLYLL then PTR <- LINK(INPTR)
                                   else if IsDBLYLL then PTR <- RLINK(INPTR)
                                   else {IsBST} PTR <- LCHILD (INPTR)
                              else PTR <- INPTR]

* * * BOOLEAN function EMPTY * * *
if IsARRAY then EMPTY <- NUMOFENTRY = 0
           else if HasHEADER AND IsCIRCULAR then EMPTY <- PTR = INPTR
                                            else EMPTY <- PTR = NIL

* * * BOOLEAN function IS (INKEY,F) * * *
if IsARRAY then IS <- F(INKEY,KEY(A[COUNTER]))
           else IS <- F(INKEY,KEY(PTR))


* * * BOOLEAN function EQUAL(X,Y) * * *
EQUAL <- X = Y


* * * BOOLEAN function GREATER(X,Y) * * *
GREATER <- X > Y

* * * procedure ADVANCE * * *
if IsARRAY then COUNTER <- COUNTER + 1
           else
             [if NOT IsDBLYLL then TRL <- PTR
              if IsSINGLYLL then PTR <- LINK(PTR)
              else if IsDBLYLL then PTR <- RLINK(PTR)
              else {IsBST} if IS(INKEY,GREATER) then PTR <- RCHILD(PTR)
                                                else PTR <- LCHILD(PTR)

* * * BOOLEAN function NOMORE * * *
if IsARRAY then NOMORE <- COUNTER > NUMOFENTRY
           else if IsCircular then NOMORE <- PTR = INPTR
                              else NOMORE <- PTR = NIL

* * * procedure GETROOMFORNEW * * *
if IsARRAY then
             [for i <- COUNTER to NUMOFENTRY do A[i+1] <- A[i]
              NUMOFENTRY <- NUMOFENTRY + 1]
           else GETNEWNODE(NEW)

* * * procedure COPYDATAFIELDSIN * * *
if IsARRAY then DATAFIELDS(A[COUNTER]) <- DATAFIELDS(INRECORD)
           else DATAFIELDS(NEW) <- DATAFIELDS(INRECORD)
```

```
* * * procedure INSERTNEWNODE * * *
if NOT HasHEADER then if INPTR = NIL then PREVIOUSLYEMPTY
                                     else if PTR=INPTR then NEWFIRSTIN
                                                       else NORMALINSERT
                 else NORMALINSERT

* * * procedure PREVIOUSLYEMPTY * * *
INPTR <- NEW
if IsSINGLYLL AND NOT IsCIRCULAR then LINK(NEW) <- NIL
else if IsSINGLYLL {AND IsCIRCULAR} then LINK(NEW) <- NEW
else if IsDBLYLL then LLINK(NEW),RLINK(NEW) <- NEW
else {IsBST} LCHILD(NEW),RCHILD(NEW) <- NIL

* * * procedure NEWFIRSTIN * * *
if IsSINGLYLL then
                [LINK(NEW) <- PTR
                 if IsCIRCULAR then
                                  [while LINK(PTR) <> INPTR do PTR <- LINK(PTR)
                                   LINK(PTR) <- NEW
 else if IsDBLYLL then
                   [RLINK(NEW) <- PTR
                    LLINK(NEW) <- LLINK(PTR)
                    LLINK(RLINK(NEW)),RLINK(LLINK(NEW)) <- NEW]
else {IsBST} LCHILD(NEW),RCHILD(NEW) <- NIL

* * * procedure NORMALINSERT * * *
if IsSINGLYLL then
                [LINK(NEW) <- LINK(TRL)
                 LINK(TRL) <- NEW]
else if IsDBLYLL then
                   [RLINK(NEW) <- PTR
                    LLINK(NEW) <- LLINK(PTR)
                    LLINK(RLINK(NEW)),RLINK(LLINK(NEW)) <- PTR]
else {IsBST}
        [LCHILD(NEW),RCHILD(NEW) <- NIL
         if IS(INKEY,GREATER) then RCHILD(TRL) <- NEW
                              else LCHILD(TRL) <- NEW]

* * * procedure DELETEifFOUND * * *
TRAVtoFINDWHERE
if NOT FOUND then
               [if NOT IsARRAY then DELETEOLDNODE
                COPYDATAFIELDSOUT
                RETURNROOMFOROLD

* * * DELETEOLDNODE * * *
    if NOT HasHEADER then if PTR = INPTR then OLDFIRSTOUT
                                         else NORMALDELETE
                     else NORMALDELETE

* * * OLDFIRSTOUT * * *
if IsSINGLYLL AND NOT IsCIRCULAR then INPTR <- LINK(PTR)
else if IsSINGLYLL {AND IsCIRCULAR} then
          if LINK(PTR) = PTR then INPTR <- NIL {Newly Empty}
                            else [INPTR <- LINK(PTR)
                                  Y <- INPTR
                                  while LINK(Y) <> PTR do Y <- LINK(Y)
                                  LINK(Y) <- INPTR]
else if IsDBLYLL then
          if RLINK(PTR) = PTR then INPTR <- NIL {Newly Empty}
                            else [INPTR <- RLINK(PTR)
                                  LLINK(RLINK(PTR) <- LLINK(PTR)
                                  RLINK(LLINK(PTR) <-RLINK(PTR)]
else {IsBST} ...

* * * NORMALDELETE * * *
if IsSINGLYLL then LINK(TRL) <- LINK(PTR)
else if IsDBLYLL then [RLINK(LLINK(PTR) <- RLINK(PTR)
                       LLINK(RLINK(PTR) <- LLINK(PTR)]
else {IsBST} ...
```

```
* * * COPYDATAFIELDSOUT * * *
if IsARRAY then DATAFIELDS(OUTRECORD) <- DATAFIELDS(A[COUNTER])
          else DATAFIELDS(OUTRECORD) <- DATAFIELDS(PTR)

* * * RETURNROOMFOROLD * * *
if IsARRAY then
             [for I <- COUNTER + 1 to NUMOFENTRY do A[I-1] <- A[I]
              NUMOFENTRY <- NUMOFENTRY - 1]
          else RETURNNODE(PTR)
```



REFERENCES

[1] Carbonell, Jaime, "Derivational Analogy in Problem Solving and Knowledge Acquisition," Proc. of the International Machine Learning Workshop(ML-83), pp. 12-18.

[2] Carbonell, Jaime, "Learning by Analogy: Formulating and Generalizing Plans from Past Experience," in Machine Learning, An Artificial Intelligence Approach, R.S. Michalski, J.G. Carbonell and T.M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983.

[3] Mayer, R.E., "The Psychology of How Novices Learn Computer Programs," ACM Computing Surveys, 13:1 1981.

[4] McDermott, D.V., "Planning and Acting," Cognitive Science, 2:2 1977.

[5] Perlis, Alan, "Epigrams on Programming," ACM SIGPLAN, 17:9 1982.

[6] Peterson, Gerald, "Using Generalized Programs in the Teaching of Computer Science," ACM SIGCSE Bulletin, 15:1 1983.

[7] Soloway, E. and Woolf, B., "Problems, Plans, and Programs," ACM SIGCSE Bulletin, 12:1 1981.

[8] Wilensky, R., Planning and Understanding, Addison Wesley, Reading, MA, 1983.