Manipulating Dynamic Data Structures with Pictures

We present a set of concepts which, together, allow pictorial programming of dynamic data structure manipulations.

The work proceeded from a desire to improve upon the "variable_name, up-arrow dot fieldname is assigned value" paradigm for manipulating pointer-based data structures. This paradigm is sufficient and compact, but it forces programmers to work at such a detailed level that they lose sight of the big picture.

Most data-structure texts present this desirable abstract point of view by showing pictorial representations of dynamic data structure manipulations. However, these diagrams are big, and they take a long time to draw. Consequently, there is usually only a single, general-case diagram, and the reader is required to construct mental models of the other cases by extrapolating from the general case, sometimes with the help of a textual description, and sometimes without.

Some workers (refs!) have approached the problem by providing an inferencing system based on a single general-case picture, but there is rarely enough information in a general-case diagram to define all other operations unambiguously.

The system described here attempts a less ambitious feat. It returns to the conventional style of programming, in which all possible cases are enumerated, but provides several mechanisms to reduce the complexity of generating these cases.

- Data structure declarations are created by the use of a simple drawing tool, which allows the programmer to visualise a graphic representation appropriate to the structure.
• Using the diagrams created in the declarations as templates, the system can automatically draw the components of diagrams showing the data structure in a particular configuration.

• Manipulations can be represented as before and after picture-pairs. If a data structure matches the configuration in the before-picture, it is altered to produce the configuration in the associated after-picture.

• Manipulations of diagrammatic components common to both pictures in a BAPP (before/after picture pair) are propagated both pictures. For example, if a pair of nodes in an after-picture have to be moved apart to allow another node to be inserted between them, the corresponding nodes in the before-picture are moved to the same configuration.

• A movable, transparent window shows the part of the data structure which is currently being worked on. The window is an alternative way of presenting the same information as is available via the conventional current_node pointer, and moving it is an alternative to updating such a pointer by a statement such as current_node := current_node^.next.

• After the programmer has supervised the drawing of a single, general case BAPP, the system can generate most of the other before-pictures automatically, and the programmer can supply the necessary information to fill out the other after-pictures.

**Diagrammatic declarations**

Data structure declarations are created by the use of a simple drawing tool. When a variable is declared to be of type pointer, the interface provides the programmer with a simple set of drawing tools with which to determine the structure of the node the pointer references (if the pointer is to be used to point at a dynamic data structure - simple types are also available). The tool will allow the creation of contiguous rectangular areas, in any two-dimensional relationship to each other. Thus a simple binary tree node would be declared as:

![Diagram of a binary tree node]

where the dashed headless arrows represent pointers to other nodes of the same type.
Further, a reshaping tool will allow the user, by creating and deleting corners, to change the rectangles into polygons of any shape. Thus, to take a particularly complex example, if the programmer were generating a data structure to represent a surface tessellated with rectangles:

![Diagram of a data structure](image)

and needed to be able to access all the neighbours of any rectangle, then we could use a data structure declared thus:

![Diagram of a node with fields](image)

The large node contains eight fields, the four in the central square containing its maxima and minima in the x- and y-directions, and the outer four containing pointers to lists of nodes which can point to other nodes like the big one.

The gray node in the tessellation below, with four neighbours above, two to the right, two below, and two to the left, would therefore be represented by a node in the data structure like the one shown to its right:
Automatic diagram generation

Once diagrammatic declarations are in place, it is possible for a programmer to describe a particular configuration of a data structure by getting the system to draw the required nodes automatically. This would involve positioning a pointer variable (which would automatically be accompanied by an arrowtail) of the appropriate type on the screen and dragging from the arrowtail. The arrowtail would lengthen into an arrow, and where the programmer stopped dragging it, a node of the type referenced by the pointer would appear. The programmer could delete the node to make the pointer a nil pointer, or could fill in the node’s data fields, and extend its pointers in a similar fashion to the first, to create a diagram showing some more complex arrangement of nodes.

This is the first step toward a paradigm which allows tests on and modifications to a specific configuration of the data structure.

Before/After Picture Pairs (BAPPs)

In order to program data structure manipulations intuitively, we need to be able to develop a formal syntax for showing the manipulations. The common element in most pictorial representations is that they allow us to show the state of the data structure, or part of it, both before and after a manipulation has occurred. Sometimes a single diagram is used, and the before information is distinguished from the after information by a graphical convention, such as a different hue or intensity, or a distinction between solid and dashed outlines or pointers. Sometimes there are explicit before and after pictures, and so on.

We have chosen the following convention:

• manipulations are represented by a pairs of pictures. A before picture shows the status of the data structure before the manipulation, and is compiled into a test for a particular
configuration of the data structure. The test comprises a check that all the nodes included in the picture exist, and are linked as the picture shows, and a check to determine whether the values or value ranges shown in the data fields are consistent with the values in those data fields in memory.

- data structure may be shown in whole or in part. The latter case is represented by a clear window onto the part of the data structure under consideration, surrounded by a gray field representing the rest of the structure.

- arrowtails in pointer fields represent pointers with unspecified values (This provides a transparent way of copying both nil and non-nil pointers), and the programmer can convert an arrowtail into an arrow by positioning the cursor on it and dragging thence to the position of the arrowhead. At the position where the drag ends, a node of the type referenced by the pointer appears. The programmer can delete the node to convert the pointer to a nil pointer, or can select and delete the pointer to convert it back to an unspecified pointer.

- generation of a data structure picture starts with the programmer typing the data structure's entry pointer in the before picture, and dragging it out to form a node.

- any pointer may be converted from a direct pointer to an indirect pointer. If this is done, the gray box with a window in it appears around the part of the structure referenced by the pointer, and the pointer becomes dashed where it passes through the gray box. (The interface emits an error message if this is done to a link in a circular structure?)

- a before picture may contain more than one data structure or more than one window onto a single structure.

- A before-picture has a broad arrowhead on the boundary adjacent to the after-picture, pointing at the after-picture.

- An identical after-picture is automatically created when a before-picture is created.

- Editing the after-picture is similar to editing the before-picture, except that the programmer may attach a broad arrow to a node in an after_picture, to indicate that it is to be disposed of, and the window in a partial view may be moved, to indicate that the part of the structure being considered is to be updated. This is a single-picture equivalent to updating pointers in a conventional HLL.
e.g.  previous_node := current_node;
current_node := next_node;
next_node := next_node^.next

and allows for structure traversal loops

• New nodes may be added to an after-picture. They are not added to the associated before-picture.

• Nodes may be shifted in and after-picture. They are automatically shifted the same amount in the before picture. Pointer connections rubberband in both pictures. For example, if a pair of nodes in an after-picture have to be moved apart to allow another node to be inserted between them, the corresponding nodes in the before-picture are moved to the same configuration.

• after-pictures may also contain assignments and procedure calls, and before-pictures may also contain Boolean expressions.

A before-picture is compiled into a sequence of tests which determine whether or not the data structures it shows are accurately described by the picture. If the tests are all successful, the new configuration shown in the after-picture is generated.

The ButWhatDoIDolf algorithm

After the programmer has supervised the drawing of a single, general case BAPP, the system can generate most of the other before-pictures automatically, and the programmer can supply the necessary information to fill out the other after-pictures.

The user presents the system with the general case of a data structure manipulation. For example, an insertion into a linked list can be presented as:

\[
\begin{array}{c}
\text{L} \downarrow \\
\text{x} \downarrow \\
\text{x} \downarrow \\
\text{x} \downarrow \\
\end{array}
\]
On seeing this, the system can start dreaming up other configurations in which the data structure can exist, and starts presenting them to the user. E.g.

\[ L_1 \]

means the structure traversal terminates before anything resembling the desired pattern is detected.

\[ L_1 \]

\[ \langle x \rangle \]

means that the first node is present, but the second node is not.

Note that in the diagrams below, instead of using an explicit current pointer, I have experimented with an unnamed current window, as a more intuitive way of representing "the bit of the data structure we're interested in". The adoption rejection of this suggestion is independent of the adoption or rejection of the WhatDoIDoIf algorithm. I hope the window idea is self-explanatory.

Further, in the diagram below I have used pointers with tails to facilitate the diagrammatic representation of copying a pointer without knowing whether it is pointing at a node or at nil, and I have adopted Mark's suggestion of making the before-picture change shape when the after-picture is edited to maintain the correspondence between the unaltered parts of the diagram.

The diagram below is the first pair of before and after pictures. These are very easy to draw. Nodes are instantiated by the drag-and-pop method (the head of a pointer is dragged to a position, and a menu appears there which allows the user to specify what the pointer is pointing at. In this case it can only be null or a node with a defined appearance, so the system can draw it), and the programmer can fill in the contents of the fields as required.
We need to be systematic about generating the cases. Assuming the system has some way of generating a current pointer, we can characterise the before picture above as a sequence of tests for

- a non-null current pointer
  - the node current\(^\) has a value \(\lt x\) in its data field (in general, would it be sensible to deal with all of a node's data fields in a single boolean expression?)
- a non-null current\(^.^\).next pointer
  - the node current\(^.^\).next\(^\) has a value \(\geq x\) in its data field

Using this scheme, the system could convert the diagram into a complex conditional test, comprising a set of nested if-statements. Because execution of the later tests would depend on truth of the earlier ones, checking for the situation described in the before picture could be performed without the danger of trying to access non-existent nodes in the data structure. Equally importantly, the nested if structure would provide a series of else clauses. The conditions governing execution of each of these else clauses would each correspond to a possible before-picture.

The system would automatically generate each of these alternative before-pictures, and would prompt the user to supply an appropriate after-picture. One or more of the before-/after-picture pairs would be the specification of how to traverse the data structure\(^1\)

---

\(^1\)Could it try to do this automatically? If it does, could we provide it with information which it could use to help it direct its search? For example, in a search of an ordered list, we know to stop if the data value is less than (or greater than, depending on the ordering function) the value we're looking for. Can information of this sort be incorporated into the algorithm? Would an algorithm making use of such information need to use AI techniques? Would it be deterministic? Would it run in an acceptable time?
The aim of generating all these pictures is to produce a piece of code describing the operation on the data structure. This has the form of a loop, defining a structure traversal. Each iteration of the loop may specify the section of the data structure to be examined on the next iteration, may terminate the traversal because the desired situation has been found, or may terminate the traversal because the desired situation cannot occur.

Here's what the loop looks like for the list insertion procedure, after the programmer has generated one pair of before/after pictures.

```plaintext
current := L;  { maintain a current pointer to }  
               { the item of interest in the list }  
status := searching 
while status = searching do 
  if current <> nil 
    then if current^.data < x 
       then if current^.next <> nil 
          then if current^.next^.data >= x 
              then succeed (after_picture_1, 
                           status) 
              else {A} 
          else {B} 
    else {C} 
  else {D} 

The procedure succeed alters the data structure to correspond to after_picture_1, and updates the variable status to have the value succeeded. Two similar procedures, fail and continue, would also be available. Both could alter the data structure to correspond to the after_picture passed as their first parameter, but continue would not alter the value of searching. (Actually, a single procedure called terminate might replace both succeed and fail; However, I feel that the distinction between succeed and fail may turn out to be necessary. I also recognise that I haven't specified any way in which the system can distinguish between them automatically).

Reduced to a single Boolean expression, the condition governing entry to clause A is:

```
current <> nil and current^.data < x and current^.next <> nil and current^.next^.data < x,
```

so the system generates this before picture:
If the unspecified pointer at the bottom of the diagram is nil, then a new end-node containing $x$ needs to be inserted immediately after the bottom node. If the pointer is non-null, then the data structure needs to be traversed further to find the insertion-point. The automatically-generated before-picture is therefore not specific enough, so the programmer specialises it to produce the first before picture in the diagram below (which also shows the associated move-to-the-next-node after-picture drawn by the programmer), and the system automatically produces the second before-picture (again the associated add-the new-data-at-the-end-of-the-list after-picture drawn by the programmer is shown).

case {A} therefore corresponds to these two before-after picture pairs
As a result of this piece of programming, the algorithm becomes:

current := L;
status := searching
while status = searching do
    if current <> nil
        then if current^.data < x
            then if current^.next <> nil
                then if current^.next.data >= x
                    then succeed(after_picture_1,
                        status)
                else if current^.next^.next <> nil
                    then continue(after_picture_2a)
                else succeed(after_picture_2b,
                    status)
        else {B}
            else {C}
            else {D}

  case {B} is straightforward - the before-picture has this condition:

        current<> nil and current^.data<x and current^.next = nil

  and the corresponding before-after picture pair is:

  ![Diagram](image1)

  Case {C} is another automatically-produced diagram which needs specialisation, so the programmer edits it to produce the first of the two before-pictures below; the second can then be produced automatically:
Note however, that a simple negation of the second condition used in before-picture 4a would put ≠ x into the data field of before-picture 4b, so, in producing this diagram, the system has taken into account the fact that before picture 4 had already restricted the value of the data field to values >=x.

Generation of the before-after picture pair corresponding to case {E} is straightforward

After all these have been created, the algorithm looks like this:

```
current := L;
status := searching
while status = searching do
  if current <> nil
    then if current^.data < x
      then if current^.next <> nil
        then if current^.next.data >= x
          then succeed (after_picture_1, status)
        else if current^.next^.next <> nil
          then continue(after_picture_2)
    else if current^.next^.next <> nil
      then continue(after_picture_2)
```

```
else succeed(after_picture_2a, status)
else succeed(after_picture_3, status)
else if current^.data = x
then fail(after_picture_4a, status)
else succeed (after_picture_4b, status)
else succeed (after_picture_5, status)

Note that the null list is a special case of the last diagram above (with after_picture_5). There would be situations in which one would like to treat the null list as a special case, so either the programmer would be able to pull the window up to the top of the grey area, making the pointer a solid line, or the programmer could delete the grey area entirely, leaving only the data structure, and the pointer pointing directly to it.

In the above example, the particularly clever programmer may notice that before_pictures 4a and 5 have identical after_pictures and may wish to arrange them thus:

I suppose a corresponding modification to the algorithm would need to be made. This is left as an exercise for the reader...