MULTIPLE POINTS OF VIEW IN MODELLING PROGRAMS

Charles Rich
Artificial Intelligence Laboratory
Massachusetts Institute of Technology

An important issue discussed at the workshop on data abstraction is the need to represent multiple, overlapping points of view and relationships between them. The first two sections of this paper motivate the use of multiple and overlapping points of view in modelling programs and data structures. The final section of the paper briefly describes a new formalism, called ***overlays***, which has been developed for this purpose, and which may be of general interest.

The overall goal of the research from which this paper is drawn is to compile a ***library*** of standard, generally used data and control abstractions to be used in the interactive analysis, synthesis and verification of programs. Multiple points of view were needed in this context is in order to decompose the data and control structures of users' programs in a way which makes explicit their relationship to the library of standard abstractions.

Multiple Points of View

For example, in the area of data structures, I have found it useful to view a single structure as an instance of several different abstractions, each of which emphasizes a different generalization of its structure. For example, a Lisp list can be viewed alternatively as a singly recursive data structure (the tail of a list is a list), as a labelled directed graph (wherein the nodes are Lisp cells connected by the CDR relation and labelled by the CAR relation), or as a sequence (wherein the *i*th term is the CAR of the *(i-1)*th CDR.) Each of these points of view provides, via the library, its own vocabulary for specifying properties of the data structure and a standard set of manipulations involving the data structure.

A singly recursively data abstraction is defined in the libary as having two parts called Head and Tail, where the Tail is recursively defined. Two standard operations on this data abstraction are called Push and Pop. Viewing a Lisp list as a singly recursive data structure is appropriate for understanding CONS and CDR as the implementation of Push and Pop operations.

A directed graph is defined in the library as a set of nodes and an edge relation. Two standard manipulations on directed graphs are splicing in and splicing out a node. This view of a Lisp list is the natural one in which to view RPLACD as the implementation of a splicing operation.

A sequence is defined abstractly as a mapping from the natural numbers. This point of view is the most natural one in which to specify the ordering properties of a Lisp list.

The important point here is that all three points of view may be required in order to describe how a single Lisp list is used in a single program. The overlay formalism, which will be decribed later, gives a way of representing the relationship between these points of view both in the abstract and for specific instances.

Overlapping Implementations

This section presents an example of multiple points of view involving procedure and control abstractions. In this example, the relationship between an input-output specification and its implementation is treated as a point of view. As above, the motivation for this approach is to explicitly represent the relationship between a specific program and a library of standard abstractions. This example also illustrates how two such points of view may overlap.

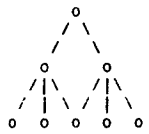
The abstract specification of the Lisp program below has two parts: to find the maximum element of a (non-empty) list, L; and to find the minimum element.

```
(PROG (MAX MIN L)
   (SETQ L ...)
   (SETQ MAX (CAR L))
   (SETQ MIN (CAR L))
   (MAPC '(LAMBDA (N)
            (COND ((> N MAX)(SETQ MAX N)))
            (COND ((< N MIN)(SETQ MIN N))))
         (CDR L))
   ...MAX...
   ...MIN...)
```

A standard implementation for finding the maximum or minimum element of a list is compiled in the library as a control abstraction with the following three essential components:

(i) an initialization,
(ii) a standard list traversal (MAPC), and
(iii) a comparison on each step between the current list element and the current max or min.

Notice that in the program above the standard list traversal component (ii) is ***shared*** between the implementation of finding the maximum and finding the minimum. This amounts to taking two different, overlapping points of view on this component, namely as part of the implementation of finding the maximum and as part of the implementation of finding the minimum. These relationships can be diagrammed as shown below.



The top node in this diagram represents the entire program. At the next level, the program is viewed as the composition of two specifications, one which finds the maximum and one which finds the minimum. The third level shows how the lowest level components are grouped and viewed as the implementation of the two abstractions above. Notice that at the lowest level there are five rather than six components -- one of them is doing double duty. This type of analysis is a violation of strictly hierarchical decomposition, which is currently the dominant technique in program design. I have in general found, however, that it is not possible to maintain strictly hierarchical analysis and at the same time capture the appropriate generalizations.

Overlays

An overlay is formally a triple made up of two programming language independent data or control abstractions, called ***plans***, and a set of ***correspondences*** between parts of the two plans. Each plan represents a point of view; the correspondences specify how information is propagated between the two views. Overlays are similar to Sussman's "slices" [2] which he uses to represent the relationship between different points of view in electronic circuit descriptions.

Fig. 1 shows an example of an overlay defined in a simplified and abbreviated diagrammatic notation. This notation will be described briefly below (see [1] for more detail and a formal definition of plans and overlays in first order logic).

The left hand side of Fig. 1 is the plan for the standard three part implementation of max/min described informally above. A plan is in general defined by a set of parts (called ***roles***), which may be operations, tests or data structures, and a set of ***constraints***, in particular data flow and control flow.

Heavy rectangular boxes in plan diagrams, such as the ones marked "initialize" and "accumulate" in Fig. 1, denote operation roles, which are specified by input-output specifications. Data flow between the inputs (top) and outputs (bottom) of operations are indicated in plan diagrams by heavy black lines between boxes. Heavy rectangular boxes with divided bottoms, such as the one marked "next" above, denote tests. The dashed lines between tests and other boxes specify the conditional flow of control. Finally, the spiral symbol in the plan diagram on the left of Fig. 1 indicates recursion. In plans, loops are represented uniformly as tail recursion.

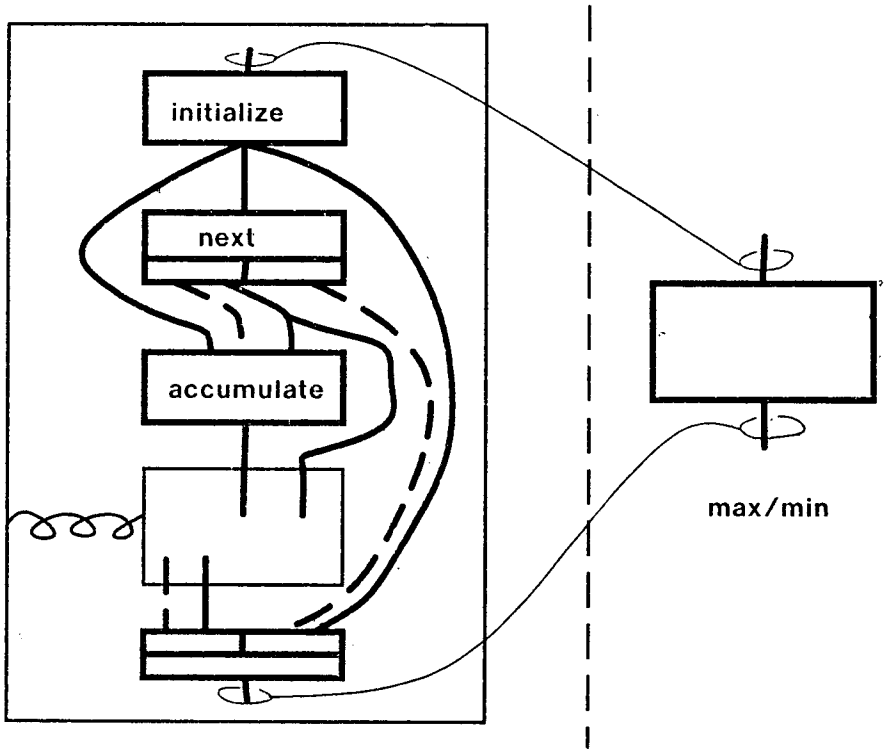


Fig. 1 - An Implementation Overlay

The box on the right hand side of the overlay diagram in Fig. 1 represents the abstract specifications for max/min (a single input-output specification is a degenerate case of a plan).

The left and right hand sides of Fig. 1 represent the implementation and specification points of view on a general programming technique. The relationship between these two points of view is expressed by the hooked lines (correspondences) between roles of the two plans. These correspondences are logically equalities. In this example, there are two correspondences: the input to the initialization of the implementation plan is the same as the input to the abstract specification; and the output at the bottom of the implementation plan (the final value of the loop) is the same as the output of the abstract specification.

Notice that overlays are ***symmetric***. This means that either side can be used as a "pattern", which makes it possible to use the same overlays in both analysis and synthesis. The fact that correspondences are equalities also means that information can propagate between points of view in both directions. For example, in bottom-up analysis, one tries to recognize known implementation plans, such as the left hand side of Fig. 1. Once an implementation plan has been recognized, it is replaced by the corresponding abstract plan and analysis continues similarly. Conversely, in synthesis one matches abstract plans and instantiates implementation plans.

A second example overlay, with two points of view at the same level of abstraction, is shown in Fig. 2 This overlay captures the relationship in general between singly recursive programs which accumulate "on the way down", such as the max/min implementation above, and those that accumulate "on the way up", such as the Lisp program below to copy a list.

```
(DEFINE COPYLIST
   (LAMBDA (L)
      (COND ((NULL L) NIL)
            (T (CONS (CAR L)
                     (COPYLIST (CDR L)))))))
```

This program is an instance of the plan on the right of Fig. 2, in which the accumulation step (in this case CONS) occurs ***after*** the recursion. The plan on the left of Fig. 2 is the same as the left hand side of Fig. 1, in which the accumulation step precedes the recursion (control flow arcs have been omitted in this overlay to make it easier to read). The four hooked lines between the two side of the overlay above specify correspondences between the two points of view. One of these correspondences is labelled "reverse". The meaning of this overlay is that, in general, accumulation on the way up can be viewed as accumulation on the way down with an intervening order reversal.

Final Remarks

The basic idea of overlays, namely using equalities between two structured descriptions to represent multiple points of view, is not limited to programs and data structures. Many other engineered devices, such as electronic circuits and mechanical assemblies can be profitably modelled as plans, with other appropriate constraints in place of data and control flow.

References

[1] Rich, C., "Inspection Methods in Programming", Ph.D. Thesis, MIT Artificial Intelligence Laboratory, June 1980.

[2] Sussman, G.J., "Slices At the Boundary Between Analysis and Synthesis", Artificial Intelligence and Pattern Recognition in Computer Aided Design, Latombe, ed., North-Holland, 1978.

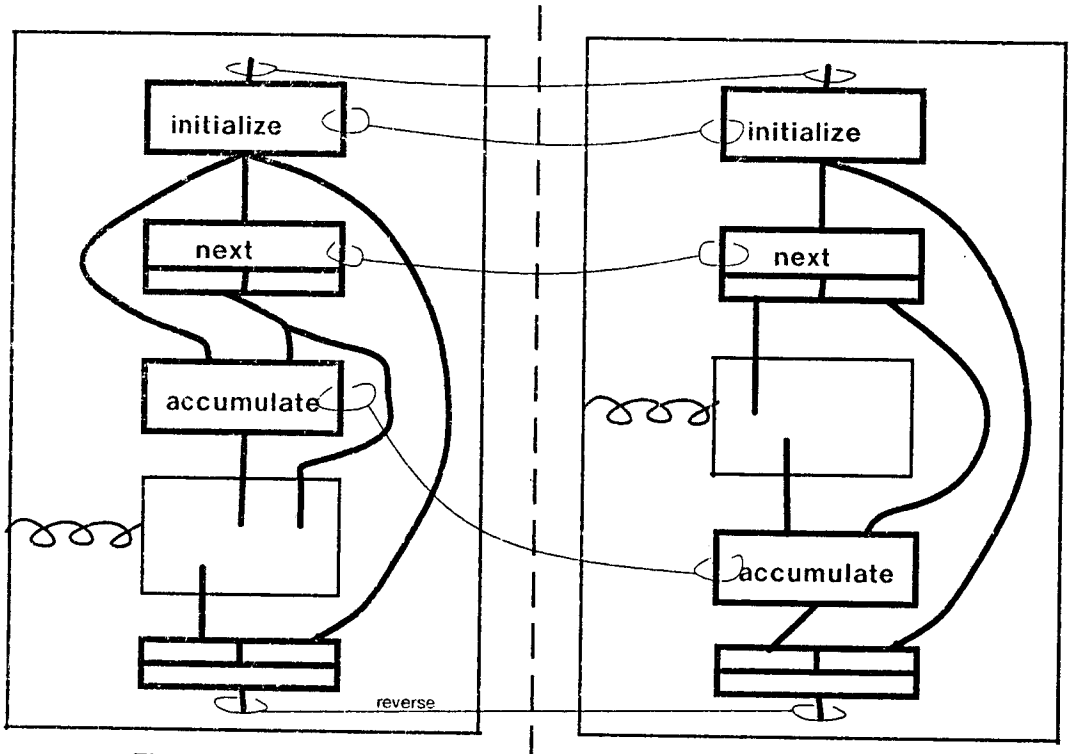


Fig. 2 - Two Points of View on Accumulation Loops