CHAPTER 8

Dynamic pointers: an abstraction for indicative manipulation

8.1 Introduction

This chapter is about manipulation, by which I mean "doing things to things". This is a rather wide definition, which at its simplest can be thought of as editing, but at its richest encompasses virtually all interaction. Most computer activities involve manipulation at several levels. For instance, you manipulate the characters and paragraphs of a file to make a document, but you also manipulate the entire document when you print it. Another example would be statistical analysis: you manipulate the individual data items to put them in suitable form, and then manipulate the whole data set with the analysis tool.

It is the opposing forefinger which enables us to be manipulative in the physical world, and of course it is our forefinger which allows us to be manipulative in the computer world.

8.1.1 Modes of manipulation

When considering manipulative operations, both in computer systems and in real life, they appear to be of two major forms:

descriptive	–	add £1500 to Alan Dix's salary
indicative	–	delete *this* line

In the former case, the *content* of the desired change is used, and in the second the word "this" would probably be augmented by a pointing finger indicating the *position* of the relevant line. Some operations involve a mixture of these two modes of manipulation, and the imperative part of the operation (the "add" or the "delete") may itself be rendered descriptively (e.g. typing) or indicatively (e.g.

clicking on a button). This leads to a strong hypothesis: *content and position are sufficient for interaction.*

To put it another way: when we want to talk about something it is in terms of *what* it is like (content) or *where* it is (position).

This hypothesis is supported in older systems where position is indicated by line numbers, file names or relative to some "current" position. However, it is even more apparent with modern mouse-based systems where all operations are supposed to be *point* and click.

8.1.2 Mediated interaction

Not only are these systems very indicative, but they have highly display-sensitive input. The meaning of *this* is the thing displayed where the mouse is pointing. We cannot describe such systems in terms of a "pipe-line" model with user's inputs being parsed, then processed and finally outputs being printed. Such a model of interaction is sufficient to describe "what you see is what you get", which relates the different outputs of the system, but it cannot handle these more manipulative systems. In these cases, we expect that any object is manipulable by indicative actions upon its displayed representation, that is: *what you see is what you can grab*. The relevant model for this is one of mediated interaction, where the current display context is central to the interpretation of the user's actions.

8.1.3 Relating levels – translating pointers

These two concepts of mediated interaction and indicative interaction are closely intertwined. In fact, from the example given, the crucial aspect of display sensitivity is knowing *what* the user is indicating. At a cruder level that means translating between the world of screen positions and the world of object pointers. All indicative systems achieve this, and the ways in which they do so are many and varied (and weird but rarely wonderful!). This relationship between levels can be represented by a pair of functions *fwd* and *back* between the pointers to the objects at the levels (*fig*. 8.1). These functions are, of course, connected to the function *proj* which represents the application objects as display objects.

This sort of translation may be repeated several times at different conceptual levels within the system. Typically there seem to be at least three levels. The innermost consists of the application objects themselves. These are then represented in some sort of virtual display, for instance, the idea of a whole formatted document, the entire drawing in a CAD system, or the collection of all windows. Finally, this unbounded virtual display is mapped into the finite physical media.

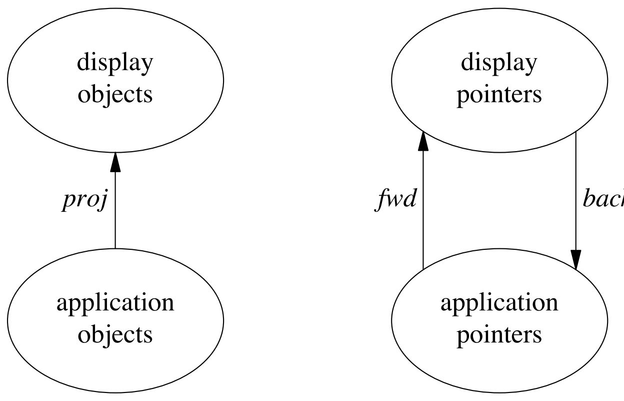


figure 8.1 *Relating levels – object and pointers*

When a user issues an indicative command, the display position can be translated using the *back* maps at each level in turn until the appropriate level for interpreting the command is reached. Similarly, if the system's response is in terms of positional information, this is translated using the *fwd* maps to appear at the screen in a suitable form.

In such a multi-layered organisation, care is needed to ensure that the repeated translations back and forth between these pointer spaces behave sensibly. One expects the *fwd* and *back* maps to be "sort of" inverses to one another. It is surprisingly hard to capture just how "sort of" this is! Further, even having decided how each level should relate back and forth sensibly, it does not follow that the composite translation behaves well. An example of this is *ladder drift*: a phenomenon where the "lexemes" implicitly defined by the pointer relations do not agree at the intermediate level. The result of this is that mapping back and forth through both levels diverges rather than stabilising.

As all indicative systems implicitly have such maps (or even several for different types of pointers), it is clearly important that they be designed, rather than come upon accidentally, in order to avoid the pitfalls described, and to present a systematic user interface.

8.1.4 Pointers in the state

The importance of positional information is clear if we consider the internal organisation of a manipulative system. The components fall into four basic classes of object:

- *Simple types* – For example, mode flags.
- *History* – For undos. This could either be abstracted into a "meta-context" or conversely be thought of as making the domain recursive. In either case

it adds no new types into the system.

- *Sub-objects* – Sub-objects of the object type that is being manipulated. For example, copy buffers, find/replace strings, etc. Strictly, these sub-objects may have some context attached to them; however, again this merely makes the domain recursive and does not add more basic types.
- *Positional information* – The cursor position, block pointers, etc.

The confidence that this list is complete stems from the above observation that manipulative actions are either of the content-oriented form "change fred to tom", or of the indicative form "do this here", or a mixture of the two. In particular, the elements that are most distinctively interactive are linked to positional information.

Further, this positional information is the major element in the interpretation of commands in relation to the display. In fact, nearly all such feedback can be regarded as simple conversion between screen position and some positional information in the underlying object.

Pointers are thus seen to be fundamental in understanding manipulative interactive systems, and we will study them in detail. We will study the way pointers change as the objects change, and thus be able to update the editing state which is constructed largely of pointers. We will be able to separate the object from its pointers as required in the previous section. We will study the relations between different objects and their pointers and thus be able to describe the transformations that occur when a command is interpreted in a particular display context, and the fidelity of these transformations.

8.1.5 Dynamic pointers

The pointers that are mentioned are not static: rather, when changes happen to the object the various pointers are changed accordingly, in the sense that they point to the same semantic entities. That is, we are interested in dynamic (semantic) pointers.

We can clarify the concept of dynamic pointers in the context of programming languages, comparing them with normal Pascal or C pointers. These are dynamic in the sense that they can be assigned different values, but are pointers to *structurally static* objects. That is, they may point to different objects and the objects they point to may change value, but the structure of the object is constant.

We want truly dynamic pointers, semantic pointers to *within* a structurally dynamic object, that change their (static) value as the object pointed to changes. A piece of pseudo-program makes this clear:

```
String   A   =  "abcde" ;
Pointer  pt  =  ref(A[4]) ;
<code to insert x at beginning of A>
print ( A, "::", val(pt) ) ;
```

The result would be :

Static pointers: `xabcde::c`

Dynamic pointers: `xabcde::d`

That is, the static pointer still points to the fourth element of `A`, but the dynamic pointer points to the fifth element of `A`: it has changed its value as interpreted statically, but is still pointing to the same semantic entity, "`d`". Examples of such pointers in other contexts include pointers to elements in linked lists (particularly simple, since in most cases the address pointed to stays fixed) and file names considered as pointers into a file system

8.1.6 How are dynamic pointers specified elsewhere?

Suzuki (1982) and Luckham and Suzuki (1979) give formal semantics for normal Pascal-type pointers; however, these are, of course, static pointers. We have to look towards user interfaces to find dynamic pointer specifications.

In Sufrin's editor, (Sufrin 1982) the cursor is very neatly handled as an intrinsic part of the object, being the separation point of the text into *before* and *after*. This leads to very clean definitions of operations at the cursor, but is not suitable for generalisation to more than one pointer. In fact, when an additional "mark" pointer is required, this must be added as a special character within the text. This is again very sensible, as on primitive display devices such a mark would have to be displayed in a character cell of its own. This is also the approach taken by Wordstar (Wordstar 1981) when allowing user markers, which are simply embedded control codes. Being an actual character in the text, such markers are guaranteed to change in a way semantically consistent with the underlying text.

The disadvantage of this approach is that the object being edited, and the pointers being used to describe the context of that editing are conflated. This is a problem even in this simple example; however, in more complex domains it becomes insuperable. For instance, the technique would be unmanageable if one allowed several editing windows on the same object, each with its own cursor and marks! In the following discussion it as crucial that the object being edited is separate from its pointers.

Dynamic pointers can be found elsewhere. Any editor with user- or system-defined marks must use them in some sense, as must hypertext systems such as NoteCards. (Halasz 1987) They may even be supplied in a programmers' interface: the SunTools graphics package for the Sun workstation has a notion of marks for user-modifiable text regions of the screen that have the properties of dynamic pointers. (Sun 1986) However, the techniques used for implementation are *ad hoc* and certainly the importance of dynamic pointers as a datatype does not seem to be recognised.

8.2 Pointer spaces and projections

We now move on to a more formal treatment of dynamic pointers. Instead of launching straight into this, we first produce a model of static pointers and then see how this can be augmented. We then go on to look at the relationships between pointer spaces using *projections*, as is necessary if we want to use pointer spaces for layered designs of interactive systems. The last two subsections describe the way that several projections can be composed, and how this affects the properties of the maps that relate the pointers at the different levels.

8.2.1 Static pointers

In the case of static pointers, clearly we have two sets of interest:

Obj	–	the set of all objects
Pt	–	the set of all possible object pointers

For example, in the simple case of character strings and pointers to these we would have:

$$\begin{aligned} Obj &= Char^{*} \\ Pt &= \mathbb{N} \end{aligned}$$

where the pointer 0 denotes before the first character, and a pointer n denotes the gap between the nth and $n+1$st character.

Of the possible static pointers, only a subset are meaningful for a particular object: for instance, the 5000th character is not very meaningful for "abc". Thus we have a valid pointers map giving the particular subset of *Pt* meaningful for a particular object:

$$vptrs : Obj \rightarrow \mathbb{P}Pt$$

So, in our example, $vptrs(\ obj\ )$ would be $\{0, \cdots, length(obj)\}$.

This valid pointers map sounds very similar to the *vhandles* map for handle spaces in Chapter 4 when we considered windowed systems; however, the meaning attached is very different. The handles are merely labels for windows and can thus be renamed at will, whereas the pointers possess a semantic meaning in themselves.

In addition, there will be operations defined on *Obj* from a set C. Each operation from C will have a signature of the form:

$$c : Params[\ Pt\ ] \times Obj \ \rightarrow\ Obj$$

$Params[X]$ represents some sort of parameter structure, including (possibly) members of the set X. In the following if f is a function $X \rightarrow Y$ then we will assume there is a canonical extension of f to $Params[X] \rightarrow Params[Y]$, which is obtained by applying f to each component belonging to X.† Similarly, if Z is a subset of X then $Params[Z]$ will represent the subset of $Params[X]$ where each component from X is constrained to lie in Z.

Example

The insert operation on strings may have a parameter structure:

$$Params_{insert}[\ Pt\ ] \quad = \quad Char \ \times\ Pt$$

with full signature:

$$insert : \ Char \ \times\ Pt \ \times\ String \ \rightarrow\ String$$

Clearly, it will only make sense to have parameters containing *valid* pointers for the object being acted on. So we will always have the precondition:

$$params, obj \in \mathbf{dom}\ c \quad \Rightarrow \quad params \in Params[\ vptrs(\ obj\ )\ ]$$

The reader familiar with type theory will recognise this as the parameter being a dependent type.

For the above example of insertion on strings, we would have for an insertion into the string "*abc*":

$$\begin{aligned} params \ \in\ & Params_{insert}[\ vptrs(\ \text{"}abc\text{"}\ )\ ] \\ &= Params_{insert}[\ \{\ 0, \cdots, length(\ \text{"}abc\text{"}\ )\ \}\ ] \\ &= Char \ \times\ \{\ 0, \cdots, length(\ \text{"}abc\text{"}\ )\ \} \end{aligned}$$

In general, the precondition requiring valid pointers is necessary but not sufficient. An operation may have a stricter precondition. For instance, consider deletion on strings.

$$Params_{delete}[\ Pt\ ] \quad = \quad Pt$$

$$delete : \ Pt \ \times\ String \ \rightarrow\ String$$

The precondition obtained from the valid pointers map would be:

† That is, *Params* is a functor on the category of sets.

$$params \in \{\, 0, \cdots, length(\, obj\,)\,\}$$

whereas in fact 0 would be an invalid pointer for deletion – there's nothing to delete. A more appropriate condition would be:

$$params \in \{\, 1, \cdots, length(\, obj\,)\,\}$$

This model describes the sort of pointer values and operations commonly encountered in programming languages. We will now extend this to allow an element of dynamism for the pointers.

8.2.2 Pointer spaces – static pointers with pull functions

The essential feature of dynamic pointers is that they change with the object being updated. The model of static pointers above is functional, and we cannot change the pointers as such (although later we will consider a model where we can). Instead we supply a function that describes the changes necessary for the pointers to retain their semantic integrity.

This is achieved by augmenting the signature of update operations:

$$c : Params[\, Pt\,] \times Obj \;\rightarrow\; Obj \times (\, Pt \rightarrow Pt\,)$$

The first part of the result is exactly as described for the static case, and we require the same consistency condition on the parameters:

$$params, obj \in \mathbf{dom}\ c \;\Rightarrow\; params \in Params[\, vptrs(\, obj\,)\,]$$

The second result from the operations is the *pull function*. This tells us how to change pointers to the original object into pointers to the updated object so that they maintain their semantic meaning. The pull function must always take valid pointers to valid pointers, and this leads to the following condition on all commands c:

$$\begin{array}{l} \forall\ params, obj \in \mathbf{dom}\ c \\ \quad c(\, params, obj\,) = obj', pull \;\Rightarrow\; pull \in vptrs(\, obj\,) \rightarrow vptrs(\, obj'\,) \end{array}$$

Example (revisited)

We can now redefine the insert operation for strings as:

$$\begin{array}{l} insert(\,(\, c, n\,), s) = s', pull \\ \text{where} \\ s' = s[1, \cdots, n] :: c :: s[n+1, \cdots, length(\, s\,)] \\ pull(\, pt\,) \;=\; pt \quad \mathbf{if}\ \ pt < n \\ \qquad\qquad\;\; = pt + 1 \quad \mathbf{if}\ \ pt \geq n \end{array}$$

Given the assumption argued at the beginning of the chapter that pointers are the principal component of editor state, we can use the pull function as a way of ensuring that this state changes "naturally" with the underlying object. For instance, if we have a set of objects *Obj* and an editing context *Context*[*Pt*], we can perform the obvious update after a change to the object $obj \in Obj$ due to an operation *c*:

$$context,\ obj \quad \rightarrow \quad pull(\ context\ ),\ obj'$$

where

$$obj,\ pull\ =\ c(\ params,\ obj\ )$$

In particular, we can use *pull* to provide our complementary function for global commands that we were looking for in Chapter 2. We could, in fact, now give a more generous definition of a global command as any operation where the parameter contains no pointers.

8.2.3 Relationships between pointer spaces – projections

As well as consistency for pointers, we also entered this chapter at an impasse when considering layered editor design. In particular, we wanted to relate the outer display layer to the underlying objects. Further, we wanted the display context to affect the parsing of user inputs, an important instant being mouse commands.

In order to describe such layering we will examine relationships between pointer spaces. In addition, this will enable us to describe one pointer space in terms of another, perhaps a simpler one, or one possessing an efficient implementation.

Any relation will be governed by a structure containing pointers and other data types similar to the parameter for operations, and we will call the set of such structures *Proj_struct*[*Pt*]. The relation, which we will call a *projection*, is a function with three results:

$$proj\colon Obj \times Proj\_struct[\ Pt\ ] \ \rightarrow\ Obj' \times (\ Pt \rightarrow Pt'\ ) \times (\ Pt' \rightarrow Pt\ )$$

The first component of the result is the simple relation between objects; the second and third are the *forward* and *back* maps, respectively. These maps relate the pointers of the two objects in a way which is intended to represent the semantic relationship between parts of the objects. Clearly, the back and forward maps are defined only for valid pointers and only valid pointers are allowed in *Proj_struct*[*Pt*]:

$$\begin{array}{l} \forall\ obj,\ proj\_struct \in \mathbf{dom}\ proj \\ \quad proj\_struct \ \in\ Proj\_struct[\ vptrs(\ obj\ )\ ] \\ \quad proj(\ obj,\ proj\_struct\ )\ =\ obj',fwd,back \quad \Rightarrow \\ \qquad\qquad fwd\ \in\ vptrs(\ obj\ ) \rightarrow vptrs(\ obj'\ ) \\ \qquad\qquad back\ \in\ vptrs(\ obj'\ ) \rightarrow vptrs(\ obj\ ) \end{array}$$

If the projection is used to describe a layered system, the target of the projection, *Obj'*, would be "closer" to the interface and *Obj* would be more an internal object. The system would be built by composing the inner object with the projection. That is, we would be interested in the behaviour of a pair: *Proj_struct*[*Pt*] × *Obj*. We can extend any command *c* on *Obj* to an operation on *Proj_struct*[*Pt*] × *Obj* with a pull function on *Pt'*:

$$\begin{array}{l} c_{proj}:\ Param[\ Pt\ ] \times (\ Proj\_struct[\ Pt\ ] \times Obj\ ) \\ \qquad\qquad\qquad \rightarrow\ (\ Proj\_struct[\ Pt\ ] \times Obj\ ) \times (\ Pt' \rightarrow Pt'\ ) \end{array}$$

$$c_{proj}(\ param,\ (\ proj\_struct,\ obj\ )\ )\ =\ (\ proj\_struct',\ obj'\ ),\ pull'$$

where

$$\begin{array}{lcl} proj\_struct' & = & pull(\ proj\_struct\ ) \\ obj',\ pull & = & c(\ params,\ obj\ ) \\ pull' & = & fwd' \circ pull \circ back \\ fwd' & = & proj(\ proj\_struct',\ obj'\ ).fwd \\ back & = & proj(\ proj\_struct,\ obj\ ).back \end{array}$$

Predictability demands that a user of such a layered system can infer the relevant internal state. If this state is of the form above, we must ask what can be inferred about *Proj_struct*[*Pt*] × *Obj* from *Obj'*, and whether any additional user-level pointers from *Pt'* are needed to determine it.

If there were a one-to-one correspondence between *Proj_struct*[*Pt*] × *Obj* and *Obj'*, this would give a pointer space structure directly to it. Usually, but not necessarily, this happens when *Proj_struct* does not have any pointer components. An example of such a mapping would be certain kinds of pretty printing, when there is a corresponding parsing function. *Proj_struct* would then contain information such as line width.

In other cases it will not even be possible to maintain a one-to-one correspondence with *Proj_struct*[*Pt'*] × *Obj'*: for instance, in a display frame map where only a portion of the underlying object is visible, in a folding map where the folded information is hidden, or in a pretty printer that displays tabs as spaces. Thus, if we use projections to model editors, observability will depend on a strategy of passive actions, as with the red-PIE. In this case we have a much better definition of passivity, namely an action that alters the *Proj_struct*[*Pt*] of the state but not the *Obj* one. In the first two cases, of the display and the folding map, we would be particularly interested in issues of faithfulness between *Obj* and *Obj′*. This is made easy by the separation of the projection into object and pointer parts. In the third example, the pretty printing, we would be interested in the existence of a single projection that gave a one-to-one correspondence between *Obj* and *Obj′*: for instance, a special view showing tabs as a special font character.

8.2.4 Composing projections

One use of projections is to aid in the layering of certain types of specification and modularisation. Bearing in mind that it will almost certainly be necessary to add extra state at each level as well as the projection information, it seams a good idea to study the composition of projections in isolation.

We will consider two projections, *proj* from *Obj* to *Obj′*, and *proj′* from *Obj′* to *Obj″*. We consider initially triples of the form:

$$Obj \times Proj\_struct[\ Pt\ ] \times Proj\_struct'[\ Pt'\ ]$$

We can then derive:

$$\begin{aligned} proj^{\dagger}:\ & Obj \times Proj\_struct[\ Pt\ ] \times Proj\_struct'[\ Pt'\ ] \\ & \qquad \to\ Obj'' \times (\ Pt \to Pt''\ ) \times (\ Pt'' \to Pt\ ) \end{aligned}$$

$$proj^{\dagger}(\ obj, ps, ps'\ )\ =\ obj'', fwd'', back''$$

where

$$\begin{aligned} fwd'' &= fwd' \circ \hat{}fwd \\ back'' &= back \circ back' \\ obj'', fwd', back' &= proj'(\ ps', obj'\ ) \\ obj', fwd, back &= proj(\ ps, obj\ ) \end{aligned}$$

This is not a satisfactory projection, as it contains pointers from *Pt′* as well as *Pt*. However, if the subset of *vptrs′*(*obj′*) given by *fwd*(*vptrs*(*obj*)) is sufficiently rich, we may not need the *Pt′* pointers at all. Instead, we can project forward to obtain *Proj_struct′*[*Pt′*] from *Proj_struct′*[*Pt*]. We can then define a proper projection *proj″* from *Obj* to *Obj″*, with projection structure *Proj_struct″*[*Pt*]:

$$Proj\_struct''[\ Pt\ ]\ =\ Proj\_struct[\ Pt\ ] \times Proj\_struct'[\ Pt\ ]$$

$$proj''\ :\ Obj \times Proj\_struct''[\ Pt\ ]\ \to\ Obj'' \times (\ Pt \to Pt''\ ) \times (\ Pt'' \to Pt\ )$$

$$proj''(\ obj, (\ ps, ps'\ )\ )\ =\ obj'', fwd'', back''$$

where

$$\begin{aligned} fwd'' &= fwd' \circ fwd \\ back'' &= back \circ back' \\ obj'', fwd', back' &= proj'(\ fwd(\ ps'\ ),\ obj'\ ) \\ obj', fwd, back &= proj(\ ps,\ obj\ ) \end{aligned}$$

This is identical to $proj^{\dagger}$ except for the term $fwd(\ ps'\ )$, to change the *Obj* pointers to *Obj'* pointers in *Proj_struct'*.

Compositions of projections are particularly useful when we want to consider multi-layer models of interactive systems. If the projection information is being embedded into the state of an editor, then the first form may be sufficient. However, as well as its theoretical interest, the second form has the advantage of only using one set of pointers, so its behaviour is likely to be more predictable and comprehensible to the user. However, any attempt to update the second structure using pointers from the surface *Pt*, will involve a lot of *back*ing and *fwd*ing. This may lead to odd behaviour unless the two maps have the right sort of inverse relationship.

8.2.5 Relation of back and forward maps and ladder drift

We have said that *fwd* and *back* relate the pointers and we expect them to be "sort of" inverses. There are several possibilities:

(i) Total inverses – the pointers are in one-to-one correspondence, rarely true.

(ii) The *back* map is an inverse of *fwd* – often true of pretty printing, but not of display-type maps where many pointers from the object are mapped to the display boundaries or bottom.

(iii) The *fwd* map is an inverse of *back* – opposite to (ii), ok for display-type maps, but no good for pretty printing where many pointers in the printed version may correspond to one in the object, for instance when padding with spaces.

(iv) There is a subset of "stable" pointers on each side of the projection in one-to-one correspondence, and *fwd* and *back* map the entire set of pointers into these "stable" subsets. This is represented by the two conditions:

(iv_a) $fwd \circ back \circ fwd = fwd$

(iv_b) $back \circ fwd \circ back = back$

Either of (ii) or (iii) implies both these conditions.

Condition (iv) seems a general one that could be demanded of all projections. Amongst other things, it imputes a lexical structure to the pointers, the set of pointers mapped together by *fwd* or *back* representing a lexeme. It does, however, suffer from incomposability.

If some property is useful, and we want to use projection compositions to modularise our design of systems, then it is important that the *fwd* and *back* maps from the different projections compose properly.

Pairs of functions, f and b say, obeying (i), (ii) or (iii) all preserve these properties when composed with similar pairs; property (iv) is unfortunately not preserved. We can summarise these properties of compositionality in a table. We have two pairs of functions (f: $Pt \rightarrow Pt'$, b: $Pt' \rightarrow Pr$) and (f': $Pt' \rightarrow Pt''$, b': $Pt'' \rightarrow Pt'$). We compose these to give $F = f' \circ f$, $B = b \circ b'$ (*fig*. 8.2).

	(ii) $b \circ f = id_{Pt}$	(iii) $f \circ b = id_{Pt}{}'$	(iv$_a$) $f \circ b \circ f = f$	(iv$_b$) $b \circ f \circ b = b$
(ii)	$b' \circ f' = id_{Pt'}$	$B \circ F = id_{Pt}$	$FBF = F \,\&\, BFB = B$**	$FBF = F$
(iii)	$f' \circ b' = id_{Pt''}$	*	$F \circ B = id_{Pt''}$	–
(iv$_a$)	$f' \circ b' \circ f' = f'$	–	$BFB = B$	–
(iv$_b$)	$b' \circ f' \circ b' = b'$	–	$BFB = B$	–n

figure 8.2 *properties of composite projections*

Unfortunately, the condition marked with the single asterisk is the one most commonly encountered in the design of display editors; an object is pretty printed to an intermediate object of infinite extent which is then viewed by a display mapping. *No* general conclusions about the resulting projection can be made, and we have to be careful in the design process to avoid problems.

The difficulty is that the lexemes generated by the two projections do not agree on the intermediate pointers, Pt'. We can see this by way of an example:

$$Pt = Pt' = Pt'' = \mathbb{N}$$

$$\begin{aligned} f(n) = b(n) &= n \quad &&\textbf{if } n \text{ odd} \quad &&\textbf{or} \quad n = 0 \\ &= n - 1 \quad &&\textbf{if } n \text{ even} \quad &&\textbf{and} \quad n \neq 0 \end{aligned}$$

$$\begin{aligned} f'(n) = b'(n) &= n \quad &&\textbf{if } n \text{ even} \\ &= n - 1 \quad &&\textbf{if } n \text{ odd} \end{aligned}$$

$\forall\, n > 0 \quad F \circ B(n) = n - 1$
{ the only stable point is 0 }

We can can call this problem *ladder drift*. We can view it pictorially as two ladders, the rungs being the stable points in one-to-one correspondence of the two projections. As we step from ladder to ladder and back we gradually slip downwards: (*fig*. 8.3).

The only way to avoid ladder drift is to have the lexemes partly "agree" on the intermediate pointers. The simplest case is when both lexemes are atomic, which is marked ** in the above table. Another alternative is to have the lexemes

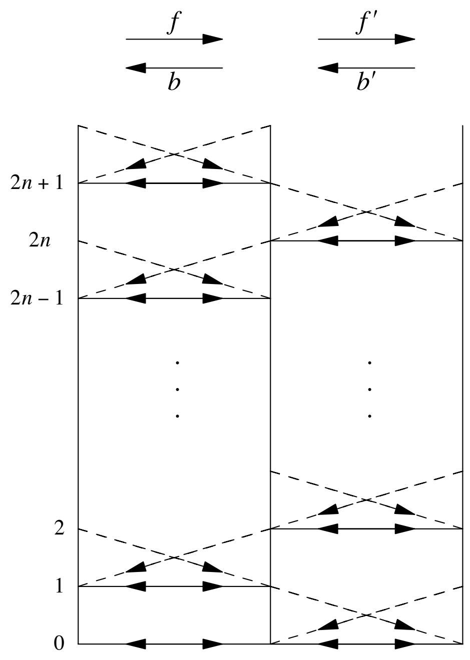


figure 8.3 *ladder drift*

generated by one projection always be complete subsets of those generated by the other. Even this lax condition is not met by pretty prints and display mappings. However, they usually satisfy it locally. Pretty prints usually have lexemes contained within lines; the display map has at its extremes two lexemes, one for everything before the screen and one for everything after, each of which consists of whole lines; in the area of the screen its lexemes are atomic. At the boundary of the two, care must be taken to ensure reasonable behaviour. Despite this complexity, it is far easier to be convinced of the reasonableness of these maps when the pointer relation is available separately from the object.

8.3 Block operations

We have already noted that pointers need not point only to positions within the object, but may also point to whole areas. Such block pointers are often used as cursors in syntax-directed editors (Bahike and Hunkel 1987, Parker and Hendley 1987).

Whereas the pull functions for operations concerning atomic pointers are fairly simple, we find that the cases of block movement and copying are far more complex. We will concentrate in this section upon block operations upon a string, partly because it is a well-known structure and the various options are easy to describe. The resulting problems are far from simple to deal with, however, and if anything linear structures have more complex behaviour and more problems associated with their block operations than do operations on tree-like data structures.

8.3.1 Block delete and insert

Before going on to the complicated examples of move and insert we will look at block delete and insert operations, which are simpler:

$$Pt = Block\_ptrs + Atomic\_ptrs$$

$$block\_delete\colon Block\_ptrs \times Obj \rightarrow Obj \times (Pt \rightarrow Pt)$$

$block\_delete((n, m), obj) = obj', pull$
where

$$\begin{aligned}
obj' &= obj[1, \cdots, n] :: obj[m+1, \cdots, length(string)] \\
pull(r) &= r && \textbf{if } r \le n \\
&= n && \textbf{if } r \in \{n+1, \cdots, m\} \\
&= r - m + n && \textbf{if } r > m \\
pull((r, s)) &= (pull(r), pull(s))
\end{aligned}$$

We might complement this with block insert:

$$block\_insert : Block\_ptrs \times String \times Obj \rightarrow Obj \times (Pt \rightarrow Pt)$$

$block\_insert(n, string, obj) = obj', pull$
where

$$\begin{aligned}
obj' &= obj[1, \cdots, n] :: string :: obj[n+1, \cdots, length(string)] \\
pull(r) &= r && \textbf{if } r < n \\
&= r + length(string) && \textbf{if } r \ge n \\
pull((r, s)) &= (pull(r), pull(s))
\end{aligned}$$

In both these cases the obvious thing to do with block pointers is to modify their endpoints using the pull function for atomic pointers. Note, however, the following behaviour for block insert:

$$
\begin{aligned}
&obj \;=\; \text{"}abcdef\text{"}\\
&b_1 \;=\; (\,1, 3\,)\\
&b_2 \;=\; (\,3, 5\,)\\
&sub\_object(\,b_1\,) \;=\; \text{"}bc\text{"}\\
&sub\_object(\,b_2\,) \;=\; \text{"}de\text{"}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\quad obj',\, pull \;=\; block\_insert(\,3,\, \text{"}xyz\text{"},\, obj\,)\\
&\textbf{then}\\
&\qquad obj' \;=\; \text{"}abcxyzdef\text{"}\\
&\qquad pull(\,b_1\,) \;=\; (\,1, 6\,)\\
&\qquad pull(\,b_2\,) \;=\; (\,6, 8\,)\\
&\qquad sub\_object(\,pull(\,b_1\,)\,) \;=\; \text{"}bcxyz\text{"}\\
&\qquad sub\_object(\,pull(\,b_2\,)\,) \;=\; \text{"}de\text{"}
\end{aligned}
$$

Is this the correct behaviour for *semantic* pointers?

8.3.2 Block move and copy

In defining these operations, we expect more problems. To help us we consider several different algebraic properties we might want to hold:

(i) $copy(\,bpt,\, pt,\, obj\,) \;=\; block\_insert(\,pt,\, sub\_object(\,bpt,\, obj\,),\, obj\,)$

(ii_a) $move(\,bpt,\, pt,\, obj\,) \;=\; obj',\, pull$
where

$$
\begin{aligned}
&pull \;=\; pull_{insert} \circ pull_{delete}\\
&obj',\, pull_{insert} \;=\; block\_insert(\,pull_{delete}(\,pt\,),\, s_{bpt},\, obj_{delete}\,)\\
&s_{bpt} \;=\; sub\_object(\,bpt,\, obj\,)\\
&obj_{delete},\, pull_{delete} \;=\; block\_delete(\,bpt,\, obj\,)
\end{aligned}
$$

(ii_b) same as (ii_a) with delete and insert swopped around.

(iii) $move(\,bpt,\, pt,\, obj\,) \;=\; obj',\, pull$
where

$$
\begin{aligned}
&pull \;=\; pull_{delete} \circ pull_{copy}\\
&obj',\, pull_{delete} \;=\; block\_delete(\,pull_{copy}(\,bpt\,),\, obj_{copy}\,)\\
&obj_{copy},\, pull_{copy} \;=\; copy(\,bpt,\, pt,\, obj\,)
\end{aligned}
$$

(iv) $copy(\,bpt,\, pt,\, obj\,) \;=\; obj',\, pull$
where

$$\begin{aligned}
&pull \;=\; pull_{insert} \circ pull_{move}\\
&obj',\, pull_{insert} \;=\; block\_insert(\; pull_{move}(\; pt \;),\, s_{bpt},\, obj_{move} \;)\\
&s_{bpt} \;=\; sub\_object(\; bpt,\, obj \;)\\
&obj_{move,}\; pull_{move} \;=\; move(\; bpt,\, pt,\, obj \;)
\end{aligned}$$

If *pt* is not contained in *bpt*, (ii$_a$) and (ii$_b$) are identical. If, however, *pt* is contained in *bpt* then (ii$_b$) would imply *move*(*bpt*, *pt*, *obj*) is the same as *delete*(*bpt*, *obj*), not quite as intended! Some systems (e.g. ded; Bornat and Thimbleby 1986) deliberately disallow this case, as it is difficult to reason about even informally and is (for similar reasons) often difficult to implement.

In both cases of (ii), the pointers from within the moved block are not moved with the block, but are destroyed – in fact, mapped to the single old block position. In conjunction with them, (iv) says that *copy*(*bpt*, *pt*, *obj*). *pull* maps all pointers within *bpt* to the old *bpt* position, even though that area has been unchanged.

Suppose now that (i) is true. In this case (ii$_b$) and (iii) are identical. Condition (iv) is inconsistent with (ii$_a$) or (ii$_b$). This is because (iv) corresponds to a "drag 'em along" policy on pointers, whereas condition (i) together with (ii$_a$) or (ii$_b$) corresponds to a "keep 'em still" policy. The latter has the great advantage of monotonicity on pointers.

The "drag 'em along" policy has further strange effects. We consider moving and copying the block (*n*,*m*) to the point *p*. For simplicity, we assume $p \le n$:

$$move(\; (\; n,\, m \;),\, p,\, obj \;) \;=\; obj',\, pull$$

where

$$\begin{aligned}
obj' \;=\; & obj[\; 1,\cdots,n \;] :: obj[\; m+1,\cdots,p \;] :: obj[\; n+1,\cdots,m \;]\\
& :: obj[\; p+1,\cdots,length(\; obj \;) \;]
\end{aligned}$$

$$\begin{aligned}
pull(\; r \;) \;&=\; r && \textbf{if } r \le n\\
&=\; r+p-m && \textbf{if } r \in \{n+1,\cdots,m\}\\
&=\; r-m+n && \textbf{if } r \in \{m+1,\cdots,p\}\\
&=\; r && \textbf{if } r > p
\end{aligned}$$

$$pull(\; (\; r,\, s \;) \;) \;=\; (\; pull(\; r \;),\, pull(\; s \;) \;) \quad \textbf{??}$$

$copy(\ (\ n,\ m\ ),\ p,\ obj\ )\ =\ obj',\ pull$

where

$$obj'\ =\ obj[\ 1,\cdots,p\ ] :: obj[\ n+1,\cdots,m\ ] :: obj[\ p+1,\cdots,length(\ obj\ )\ ]$$

$$\begin{aligned} pull(\ r\ ) &= r && \textbf{if}\ r \le p\ \textbf{and}\ r \in \{n+1,\cdots,m\} \\ &= r+p-n && \textbf{if}\ r \in \{n+1,\cdots,m\} \\ &= r+m-n && \textbf{if}\ r > p \end{aligned}$$

$$pull(\ (\ r,\ s\ )\ )\ =\ (\ pull(\ r\ ),\ pull(\ s\ )\ )\quad \textbf{??}$$

The pull for the pointers is obvious given the policy, but the pull for the blocks (marked with **??**) is not so clear. In fact, the non-monotonicity of the pull function has had disastrous consequences on the blocks, as we can see from the following example:

$obj\ =\ "abcdefg"$
$b_1\ =\ (\ 3,5\ )$
$b_2\ =\ (\ 4,6\ )$
$sub\_object(\ b_1,\ obj\ )\ =\ "de"$

let $obj',\ pull\ =\ block\_move(\ b_2,\ 2,\ obj\ )$
then
$obj'\ =\ "aefbcdg"$
$pull(\ b_1\ )\ =\ (\ 5,2\ )\ =\ (\ 2,5\ )\quad \textbf{??}$
$sub\_object(\ pull(\ b_1\ ),\ obj'\ )\ =\ "fbc"$

As we see, the block pointer gets its ends reversed. If we assume (at the point marked **??**) that this is the same as the properly ordered block pointer, then we get the very odd result for the sub-object of the pulled block. This is clearly not correct behaviour for semantic pointers.

The alternatives for the choice of block pull function are either to treat pointers like $pull(\ b_1\ )$ as invalid and say $(\ 5,2\ ) = bottom$, put up with silly behaviour like "*fbc*" above, or have more complex block pull functions. Both those blocks totally contained in *bpt* and those disjoint from it behave as we would expect. Blocks bridging either end of *bpt* could be curtailed in some way, either by removing the bit inside, or the bit outside *bpt*. There are two alternatives for this chopping. We could chop off the inside, that is, a "leave 'em behind" policy for these blocks – within the general "drag 'em along" policy for atomic pointers:

$$pull_{move}(\ (\ r,\ s\ )\ )\ =\ (\ r',\ s'\ )$$

where

```
if $r, s \in \{n+1, \cdots, m\}$ or $r, s \notin \{n+1, \cdots, m\}$
then
        $r' = pull(\ r\ )$
        $s' = pull(\ s\ )$
else
        $r' = pull(\ n\ )$   if $r \in \{n+1, \cdots, m\}$
           $= pull(\ r\ )$   otherwise
        $s' = pull(\ n\ )$   if $s \in \{n+1, \cdots, m\}$
           $= pull(\ s\ )$   otherwise
```

Alternatively, we could chop off the bit outside, leading to a "drag 'em along" policy for these blocks:

$$pull_{move}(\ (\ r, s\ )\ ) = (\ r', s'\ )$$

where

```
if $r, s \in \{n+1, \cdots, m\}$ or $r, s \notin \{n+1, \cdots, m\}$
then
        $r' = pull(\ r\ )$
        $s' = pull(\ s\ )$
else
        $r' = pull(\ n\ )$   if $r \notin \{n+1, \cdots, m\}$
           $= pull(\ r\ )$   otherwise
        $s' = pull(\ n\ )$   if $s \notin \{n+1, \cdots, m\}$
           $= pull(\ s\ )$   otherwise
```

These strategies, although possible, are hardly simple or clear and have very strange behaviour on the block inclusion relationship.

Another way round the problem is to allow complex blocks consisting of lists of intervals.† In the example given above:

$$b_1 = (\ 3, 5\ )$$
$$sub\_object(\ b_1, obj\ ) = \text{"}de\text{"}$$
$$pull(\ b_1\ ) = \{(\ 5, 6\ ), (\ 1, 2\ )\}$$
$$sub\_object(\ pull(\ b_1, obj'\ )\ ) = \{\text{"}e\text{"}, \text{"}d\text{"}\}$$

This solution still has problems, since intervals can become non-intervals; we may need to define block moves and their pulls with non-interval parameters!

† I have been told that this is the solution adopted for links in the Xanadu hypertext system (Nelson 1981), although I have not seen any documentation confirming this.

8.3.3 Block move in various editors

It is instructive to look at various editors and see how they handle pointer movement for block operations. We will consider four editors. Vi (Joy 1980) and ded (Bornat and Thimbleby 1986) are two heavily screen-based text editors operating under Unix. Wordstar, (Wordstar 1981) in various versions is probably the world's most widely used word processor for microcomputers. Spy (Collis *et al*. 1984) is a mouse-based multi-file editor for bit-map workstations.

- *Atomic pointers* – Vi inherits line pointers from its line editor predecessor; however, it uses cut/paste rather than an atomic block copy or move. It therefore follows a "leave 'em behind" strategy, satisfying conditions (i) and (ii_a). Vi gets round this very neatly, by having only one pointer, the current cursor position, and making all insertions at this point! Ded has only one atomic pointer, the cursor, and it always moves this to the site of block deletion, or movement, irrespective of where it started off. Wordstar has multiple atomic pointers, but these are represented within the text as control sequences, thus they are moved with the rest of the text in block moves, and the user can even choose whether or not to include the pointers at the block boundary. Spy has marks which are moved with the text in block moves, and *duplicated* in block copy.
- *Block pointers* – Vi doesn't have any, however, because it uses cut/paste; if it were to have block pointers they would presumably be handled by the "leave 'em behind" strategy. Ded allows only non-intersecting blocks, and hence avoids the intersecting blocks problem. Wordstar and Spy both have only one block pointer (in Spy's case the selection) and so also avoid this problem.

We see that common editors usually solve these problems in an *ad hoc* manner, sometimes deliberately avoiding them, sometimes not possessing the functionality to show them up. They also tend to incorporate many special cases (especially for the cursor/selection). These inconsistencies are precisely what we want to avoid by use of pointer spaces. Even if some inconsistency is tolerable for monolithic editors, it is not acceptable where generic components are required or where concurrent access to the edited object is required. A good rule of thumb is to adopt the "leave 'em behind" strategy, possibly with special rules for cursor/selection.

8.4 Further properties of pointer spaces

In this section we discuss briefly some more advanced properties of pointer spaces, particularly those concerned with the formal treatment of block pointers. A more rigorous treatment of these properties, many of which are simply generalisations of the properties of intervals, can be found in Dix (1987b).

8.4.1 Types of pointer and relations on pointers

So far, no structure has been assumed on the set of pointers. However, in practice most pointers spaces are highly structured. This structure includes:

- *Special elements* – e.g. beginning and end pointers.
- *Classes of pointers* – e.g. for strings before-character and after-character pointers, and for trees, node and leaf pointers.
- *Ordering and other relations* – e.g. partial or total ordering of *Pt* or *vptrs*(*obj*).

The relation on pointers is most general, classes being unary relations and special elements being classes of a single element. They can thus all be described in the same way, but there is certainly a perceived difference.

The structure of the object may force structure on the pointers: for instance, an object consisting of two major sub-objects may have a set of pointers divided into two classes according to the sub-object to which they point. Conversely, the pointers may be used to impute structure to an otherwise unstructured object: for instance, a string of characters may have pointers corresponding to paragraphs, sentences and words.

We have already considered the case of block pointers onto string; in general, there are likely to be *block pointers* indicating portions of the object. These typically have associated with them a whole set of relations of their own (e.g. inclusion, intersection), and special sub-object projection maps.

8.4.2 Absolute and relative structure

As we have seen, some structure is built into the pointers themselves, and some they inherit from the objects. When the structure is inherited from some non-constant feature of the objects, or when it is defined using the objects, it will be *relative* to a particular object. If, on the other hand, the structure can be defined independently of any particular object, we will say it is *absolute*. If $F(Pt)$ represents the structure operator we are interested in, then we have the following two definitions:

absolute structure:

$$struct \in F(\ Pt\ )$$

relative structure:

$$struct \in Obj \rightarrow F(\ Pt\ ) \quad \textbf{st} \quad struct(\ obj\ ) \in F(\ vptrs(obj)\ )$$

Note that an absolute structure is a relative structure with a constant map. Hence we may, if we wish to, deal with relative structures and include absolute structures by default.

An example of a class with relative structure is word pointers on strings: whether a block refers to a word or not depends on the specific string pointed to and not on the pointer. Perhaps a more common form of relative structure is the end pointer for strings, which depends on the length of the string. This does have the useful property that it needs to be parametrised only over *vptrs*(*obj*).

There are strong consistency and simplicity arguments for trying to deal with absolute structures as much as possible. However, as the word pointer and end pointer examples illustrate, relative structures are often natural.

8.4.3 Pull functions and structure

For consistency, it would be desirable if pull functions preserved in some sense the structures on the pointers. Taking again strings as examples, most simple manipulative actions, such as *insert* and *delete*, are monotonic on the natural pointer order. Similarly, they preserve block inclusion and end pointers. They do not, however, preserve start pointers when inserting at the string beginning. The start pointer for the original string ends up pointing after the character inserted and is therefore not preserved. We might try to get round this by adding start pointers as special objects to *Pt*, $Pt = \mathbb{N} \cup \{START\}$, but we would quickly find that adding such elements complicates the description, and hides consistency (*START* behaves like pointer 0 in all respects except update).

On the positive side, note that the start pointer is altered only when it is near to the site of the insert. That is, there is some *locality* of the update outside which the pull is consistent. This idea of locality is very important in the interactive context, as changes to an object can afford to be strange so long as they are strange only in the locality of the point of interaction, and consistent elsewhere. The simple string functions even preserve (up to locality) the relative word pointers.

8.4.4 Block structure

The fact that blocks refer to sections of the underlying object means that they have some inherent structure. We would want to say whether two block pointers *intersected*, (or *interfered* with) one another, or whether one *contained* the other. Similarly, we might want to define an operation that, given two block pointers, gave a pointer as their intersection. These relations would want to satisfy sensible lattice properties, similar to those of sets or intervals. For example, we would want the intersection of two blocks to be contained in both of them.

There are some differences between the operations on block pointers and the natural operations on intervals. We want to be conservative when dealing with blocks, so we want to say that two blocks interfere even if the intersection we return is empty. Similarly, we should not insist that the intersection of two blocks is maximal.

Depending on the structure of the underlying object, we may be able to distinguish blocks that refer to several disparate parts of the object from *intervals*. This sub-class of blocks would have additional properties, especially when there is an ordering on the object and its atomic pointers.

8.4.5 Sub-object projections

Most objects of any richness at all can be viewed in part as well as in whole. Even simple integers can be decomposed into digits or into prime factors. Sometimes we have a relatively unstructured relation between object and sub-object, such as sub-strings of strings. In other cases, the relation is stronger, for instance a sub-tree of a syntax tree corresponding to a *while* loop. If the relationship between object and sub-object is sufficiently natural there will be a relation between their pointers, and we will be able to construct a projection, the defining structure of which will typically be a single block pointer. If the pointers are not rich enough for this already, they can usually be extended to include sub-object projection structures.

In more explicit terms, we are going to associate with a pointer space, $<Obj,\ P_{Obj}>$, a sub-object pointer space, $<Sobj,\ P_{Sobj}>$, and a projection:

$$sub\_object:\ Bpt_{Obj} \times Obj \ \rightarrow\ Sobj \times (\ P_{Obj} \rightarrow P_{Sobj}\ ) \times (\ P_{Sobj},\ P_{Obj}\ )$$

The normal rules for projections apply but, because of the special nature of sub-objects, the *fwd* and *back* maps will have additional properties. In particular, we would expect a one-to-one correspondence between the sub-object pointers and some subset of the object pointers. That is, *fwd* is a left inverse to *back*:

$$\begin{array}{ll} \forall\ b \in Bpt,\ obj \in Obj & \textbf{let}\ \ sobj, fwd, back\ =\ sub\_object(\ b,\ obj\ ) \\ & \textbf{then}\ \ \ fwd \circ back\ =\ id_{vptrs(\ sobj\ )} \end{array}$$

Alternatively, we could define a sub-object projection to be any projection satisfying the above condition.

Often there will be a block pointer corresponding to the whole of an object (usually relative to *vptrs*(*obj*)) that yields the whole object. That is, the *fwd* and *back* for this sub-object are one-to-one, and the resulting projection is one-to-one when considering *Obj* as whole. If we do not possess such block pointers, we can always construct them.

It will usually be the case that we can consider sub-objects of sub-objects; that is, the sub-object projection is in fact an *auto-projection*:

$$sub\_object: < Sobj,\, P_{Sobj} > \;\rightarrow\; < Sobj,\, P_{Sobj} >$$

the original objects, *Obj*, being a subspace of *Sobj*.

We expect there to be coherence properties for the sub-object projections. If we look at a sub-object B and a sub-object of it, C, then we expect the combined sub-object projections from the object to B and then from B to C to be the same as if we had gone to C directly.

The existence of a sub-object projection automatically generates some relations on the block pointers. There is an obvious containment relation by examining the block pointers of sub-objects, and we can generate an independence relation on blocks by seeing whether all possible values of the sub-objects can occur together.

8.4.6 Intervals and locality information

Blocks, usually interval blocks, have a special role to play in providing locality information. Typically an operation changes only a very small part of an object, and outside this locality the object is unchanged and the pull function takes a particularly simple form. We can express this by having certain operations return an additional block pointer value, its locality:

$$c : Params[\ Pt\ ] \times Obj \;\rightarrow\; Obj \times (\ Pt \rightarrow Pt\ ) \times Bpt$$

We will refer to this additional component as $c(\ params, obj\ ).loc$.

This locality information has the property that for any block that doesn't intersect, $c(\ params, obj\ ).loc$ yields the same object before and after the operation; similarly, the pointers yielded by the sub-object projection are invariant under $c(\ params, obj\ ).\ pull$:

```
let     $obj', pull, loc \;=\; c(\ params, obj\ )$
if      $b \in vptrs(\ obj\ )$
and     $b$ does not intersect $loc$
then    $sub\_object(\ b, obj\ ) \;=\; sub\_object(\ pull(\ b\ ), obj'\ )$
```

We could, in fact, define an optimal locality by letting the locality of *c*(*params*, *obj*) be a minimal block satisfying the above. In practice, however, we would return a conservative estimate of the locality, rather than the optimal locality. For instance, for the overwrite operation on strings, overwriting the *n*th character will normally have a locality of $(n - 1, n)$; however, if the character overwritten is the same as the original character the optimal locality is null. In such cases it may often be better to deal with the generic locality rather than the particular one, both for computational simplicity and for ease of abstract analysis.

Locality information can be used to express user interface properties. If the locality of an operation is contained within the display then the user has seen all of the changes. This gives a more precise version of the "mistakes in local commands are obvious" principle stated in Chapter 2. We have confidence not only that we can see when there has been a change, but also that we can see *all* of the change.

8.4.7 Locality and optimisation

Locality information can also be invaluable in performing various optimisations. By knowing where changes have occurred, we can cache sub-objects and know when the caches become invalid. Further, if we know the structure of pointers we can often optimise pull functions outside the locality. For example, if we know that the locality of the string operation, *op*(21307,*The_complete_works_of_Shakespeare*), is (21306, 21309) and that *pull*(21310) = 21323, then we know we can just add 13 to all pointers greater than 21306, leave unchanged those less than 21306 and perform the function call only on pointers in the range 21306 to 21309. Such optimisation can be very important, as a function call is expensive compared with addition.

Frequently such optimisations will be possible where the operation is defined using a projection and the projection has "nice" locality properties. A *self-contained* block is the simplest such locality property. It is a block whose projection is dependent only on the sub-object it contains. An example of a self-contained block, would be a paragraph in text, which can be formatted independently of the surrounding context. If an update operation is performed and the locality of the operation does not intersect with the locality of the block, then the projection of that block is unchanged.

There is a slight complication in this definition. Whether or not a block is self-contained is usually dependent on the object into which it points. That is, self-containedness is usually a relative structure. We must therefore demand that a block is self-contained *both* before and after the operation. For example, if we deleted the line between a paragraph and the preceding one, the original paragraph would be part of a larger paragraph and hence no longer be self-contained itself.

Even where self-contained blocks exist, they may be large. If we change a character within a paragraph, it will not change the formatting of much of the paragraph. Only occasionally will it even mean reformatting to the end of the paragraph. However, not even the beginning of the paragraph will be a self-contained block, as the formatting of its last line will depend on the length of the following word.

We can cater for such cases by considering the notion of the *context* of a block. This is some information that, together with the contents of the block, is sufficient to determine the projection of the block. This context will typically be a surrounding block pointer together with some additional information. Thus if the locality of an operation does not intersect the context of a block, we do not have to update the projection of the block. This generalises the notion of self-contained block sufficiently to be useful. Indeed, block contexts were used in the optimisation of an experimental editor developed at York. (ce)

8.4.8 Locality information and general change information

The locality information, like the pull function, can be thought of as an example of change information. Similar constructs appear when considering, for example, views of objects. Again a view cannot be defined outside the context of a particular object. Thus pointers and pull functions can be seen as examples of a general phenomenon. On the other hand, I believe that many such constructs can be modelled using pointers and pulls, although sometimes it will be best to import only a subset of their functionality into the new domain.

8.4.9 Generalisations of pointer spaces

The reader will have been struck by the similarity between operations with pull functions, and projections. The former are, of course, a special case of the latter where the two objects are of the same type. The reason for not including an inverse for *pull* in the definition of all operations is that we are after a way to describe the update properties of systems, and hence this is unnecessary. We could imagine cases where it would be of help. For instance, I might change a file and want to trace back a position in the changed document to the original. We might go further and say that both update operations and projections should supply a relation, rather than functions. We could constrain this relation so that it satisfies the pseudo-inverse property of §8.2.5. Again, the reason for not doing so is that we are after a formulation that will enable a system to perform the translations automatically; a relation will yield a choice that the system would not know how to deal with.

We could use relations in two sets of circumstances, though. First, in the early specification of a pointer space or projection, we might know what class of behaviours is acceptable, but not be decided exactly which as yet. This would certainly be the case with the various options for block move and copy, where the range of possibilities is reasonably clear, but not the exact choice. The second place where relations would be useful would be where the pointers are an explicit part of the interface. For instance, it might be sensible for user-defined marks to split when the block containing them is copied, or alternatively for the user to be consulted on what to do with them. This would still be arduous if the user was not given sensible defaults, especially as the number of pointers grows (as in a shared hypertext). Perhaps the best course to take would be to supply both a relation and preferred destinations (in the form of functions consistent with the relation). The designer of a system could then choose whether to use the default supplied, to split pointers over their options, or to offer the end-user choice.

The changes that would need to be made to encompass this generalisation are fairly obvious and would serve only to clutter the exposition. For that reason this chapter (long enough already!) has followed the simple method.

Generalising in a different direction, dynamic pointers are associated with positions. This was the starting point for this chapter. In fact, the axioms given for pointer spaces are very sparse and do not constrain us to this interpretation. There are other structures that are not directly positional in themselves but have similar properties. For instance, we discuss views in the next chapter. Throughout most of that chapter we deal with static views, but many views of objects in interactive applications share many of the properties of pointer spaces. Elsewhere I have shown how dynamic views can be defined in terms of block pointers and sub-object projections. (cf) This supports the hypothesis that position is central to interactive system design. On the other hand, even if we can describe a phenomenon in terms of positional pointers, it may be better (in view of our behavioural slant) to use just that subset of the properties of pointer spaces that are needed to describe it, and apply them directly to the domain of interest.

8.5 Applications of dynamic pointers

To conclude this chapter we look at the uses of dynamic pointers, both present and future. We first concentrate on the ways that dynamic pointers are used currently in a variety of systems. Some of these have been mentioned before in the chapter, but they are gathered together for reference. After this, we consider the importance of dynamic pointers in the creation of advanced user interfaces. In particular, many of the examples concentrate on programming support environments.

8.5.1 Present use of dynamic pointers

The use of dynamic pointers which we have focussed on particularly has been the design of editors. Here, as we have already seen, are found many examples: cursors, selection regions, marked blocks, display frame boundaries, folded regions. All of these constructs must have some of the properties of dynamic pointers, and would ideally have some sort of consistency between them. Usually their semantics are regarded severally and in an *ad hoc* manner; even an informal concentration on their similarity would rationalise editor design.

Hypertext systems must again use dynamic pointers; these fall into several categories. NoteCards (Halasz *et al*. 1987) is primarily a *point-to-object* system: the text of a card can contain atomic references to other cards in the system. The references to the objects can be regarded as dynamic pointers into the card database, as well as the reference points being dynamic pointers into the card's text. NoteCards also supports what it calls global links. These are links between cards that do not reside at a particular location in the card, and could be described as *object-to-object*. This is the sort of linkage handled by conventional databases. The Brown University Intermedia system (Yankelovich *et al*. 1985) uses a *block-to-block* method where blocks in the text of a hypertext object reference other blocks. However, these references take place through intermediate objects recording information about the link, so this could be regarded as a sugared form of a *block-to-object* link. The awkward problem of how to represent block links is resolved here by indicating only the start position of a referenced block and displaying the whole block on request. This problem of display arises continually when dealing with objects with a large number of links. If the density of links is too high it will be necessary to hide those types of links not of interest, or if one is interested only in the object itself then perhaps all the links. If this is the case then it becomes even more important that the movement of these links is sensible and predictable when the object is edited. In the majority of cases the pointers in the above are implemented either by embedded codes in the objects of interest or as references to representations such as linked lists.

Databases and file systems provide a wealth of dynamic pointer examples. In a way, all file names can be thought of as dynamic pointers. They usually retain the semantic link to the files to which they point, even when those files change their content. They are a particularly simple example, because of the simplicity of typical file system structures. They fail to be true dynamic pointers, in that the file name itself has (or should have) some sort of mnemonic meaning in itself: it both denotes some of the subject matter of the file and often, by means of extension names, may denote type information. Thus the file name is in a sense an attribute of the file rather than a simple handle to it. Those who call their files names such as "jim" and "mary" come closer to the ideal of dynamic pointers! Despite this difference, there are enough similarities to make the analogy useful.

In the underlying implementation of file systems one comes closer to true dynamic pointers. For instance, in the Unix file system (Thompson and Ritchie 1978), each file is denoted by an integer, its *i-node* number. Even when the file name changes, this number remains unchanged and similarly, whereas the location on disc where a file is stored may change as it is written to and rewritten, the *i-node* remains the same.

Again, at two levels databases have dynamic pointers. At the user level these are database key values: these denote a semantic record, even when the record contents change or the database is reorganised. These keys are usually meaningful, like file names: in fact, usually more so. It is often argued that this is a shortcoming in database design: for instance, a record keyed by surname and forename might change its key value when the subject married. Some more advanced relational database designs codify this meaninglessness by supplying system-defined *surrogate* keys for each entity. (Earl *et al.* 1986) This gets close to the sort of referencing between records and their sub-records in network databases, again a place where the integrity of the semantic link must be maintained despite changes in the database values.

Finally, we could look at the example (anathema to computer scientists) of the BASIC programming language. (Kurtz 1978) In this, all statements are numbered. When, as often happens, the programmer finds that the original choice of numbering left too little space for the statements required, most systems supply a command that changes the numbering to space out the program. In the process all statements that refer to line numbers have their operands changed accordingly. That is, the number "100" in "GOTO 100" is, in fact, a dynamic pointer to the statement numbered "100"!

8.5.2 Future use of dynamic pointers in advanced applications

Once one recognises dynamic pointers as a central data type, many different applications become obvious, some being self-contained, like the editor, others requiring that the pointers be an integrated part of the environment in which the application operates.

In the previous sub-section, we noted that file names are really an attribute of file objects, rather than just a handle to them. In direct manipulation systems there is less need for all objects to be denoted by file names. Naming is still important, as the work on window managers and the problem of aliasing emphasises (§1.5.2). However, names could become more of an extended annotation of the file contents, rather than the principal method of manipulation, a job taken over by the mouse and icon. In addition, temporary objects, which have no identity of their own but have importance merely for their contents, can remain anonymous, being denoted only by their spatial arrangement on the screen. In such cases the distinction between objects and the windows and icons representing them can become slightly blurred, and it is not surprising that the model for representing windows and that for dynamic pointers are similar. It is likely that the distinction between such systems and hypertexts may also become blurred: for instance, the directory or folder would become a simple list of dynamic pointers to other objects, similar to the file-box in NoteCards (Halasz *et al.* 1987). It would be quite natural that files describing the file system (perhaps containing dependency information, such as in the Unix utility "make" (Thompson and Ritchie 1978), could be created using pointers rather than names. It is unclear whether names in such a system would be an attribute of the object or of the link; perhaps both would be allowed.

Pointer spaces and projections will be indispensable for representing concurrent editing on multiple views of the same object: for instance, when manipulating a graphical object directly and at the same time editing its representation in a graphics representation language. Similarly, one might want to edit an unformatted text and its formatted form concurrently. Such dual-representation editing overcomes some of the doubts raised about the limitations of direct manipulation for representing the full functionality desired, freeing the user to choose among the different representations as is most natural and useful.

As we've already noted, an environment where dynamic pointers are normal would lead to the development of facilities such as find/replace as general tools. Contrast this for instance with Unix, which has general-purpose find tools (grep, fgrep, egrep) and global context editors (sed, awk) at the command interface level (Thompson and Ritchie 1978), but because they are not easily interfaced at the program level their facilities are repeated throughout the different editors of the system. Obviously, for some task-specific reason a designer may want to incorporate slight variants of existing tools within new applications, but she should not be forced to do so.

Thinking of such simple text tools at the command interface level, leads nicely to considering the more complex tools of a programming environment. Multi-stage compilers already have to retain some of the notion of a *back* function to the original source in order to give sensible error messages. For instance, the "C" language pre-processor (Kernighan and Ritchie 1978) embeds control lines into the output stream so that the later stages can know from which line of which file errors originated. Other similar tools in the Unix environment do not have such facilities, so for instance using the various pre-processors to the typesetter "troff" (Ossanna 1976) means that error messages are recorded at line numbers that bear no relation to the original text. A consistent framework at the level of the environment would make all these tools easier to define and use.

When error messages are reported (assuming they refer to the correct locations) dynamic pointers are again useful. For instance, the "ded" display editor (Bornat and Thimbleby 1986) has a facility for taking the error messages from a compiler and stepping through the source to the line of each error message. Unfortunately, as the file is corrected, the line numbers make less and less sense in relation to the new file. In essence, the *pull* function has been ignored. If the underlying implementation had been in terms of dynamic pointers this failing would have been avoided: the error messages would have been parsed into dynamic pointers to the original source file. (Assuming they were not already supplied in this form by the compiler.) Then as the file changed when the errors were corrected, the dynamic pointers would have kept pace with the semantic position in the file to which they refer. The Microsoft QuickC environment, (QuickC 1988) where a compiler, editor and debugger are all designed in a consistent framework, does behave in the appropriate manner, and positional elements such as breakpoints and error lines retain their semantic position through changes. Of course, the dynamic pointers are part of the QuickC environment, rather than of the operating system or file system. Therefore, if you choose to use your own editor rather than the QuickC editor the semantic positions are lost.

Where compilation is integrated into the editing environment, dynamic pointers can aid in incremental compilation and error reporting. Alternatively, they may actually form the basis of integration of editors and compilers, regarded as separate tools within a dynamic pointer-based environment. In the next sub-section we see a detailed example of how locality information can give textual editing advantages similar to syntax-directed editing in terms of incremental parsing, and superior in some important cases. It also gives an example of how the locality information can be used to give improved error reports based on assuming the changed region is at fault, and warnings where unintentional syntax slips are suspected.

Similarly, dynamic pointers can be used for proof reuse. The correctness proof between various stages in the refinement of a formal specification could be very large. It would be tedious to have to redo the whole proof after every change even when one knew that the proofs would be nearly identical. High-level proof heuristics and plans (Bundy 1983) will, of course, save this work, but in many cases it will be possible to use the proof exactly except that the pivot points for the application of the proof steps will have changed marginally. If the pivotal expressions are represented by dynamic pointers, then the positions after the change will probably lead to a completely faultless reapplication of the proof. In the cases where this fails, it will at least remove some of the tedious work from the shoulders of either the programmer or the more intelligent theorem prover.

8.5.3 Incremental parsing and error reporting using dynamic pointers

We have considered many possible uses of dynamic pointers and we will now look in detail at their use for incremental parsing and error reporting. We could expand in a similar fashion on several of the points raised above, but the present discussion can serve as an illustration.

Syntax-directed editors can achieve a large degree of incremental semantic checking and compilation because of their knowledge about the syntactic objects changed and the extent of such changes. Often, as users are not totally happy with the syntax-directed approach for all editing, they are allowed the selection of regions for standard text editing. (Bahike and Hunkel 1987) Unfortunately, such regions have to be completely reparsed and analysed when the text editing is complete. Using the *locality* information that can be supplied by dynamic pointer operations, together with the *projection* information between the existing parse tree and original text and the *pull* function relating that to the new text, one can reduce the work needed dramatically. This information can be used with both small-scale and large-scale changes:

- *Small-scale changes* – The advantages here are similar to those of syntax-directed editing. It is possible to infer from the locality of a change to the text what corresponding section of syntax tree should be changed, and just reparse this.
- *Large-scale changes* – One of the reasons for switching out of syntax editing mode may be that the syntactic primatives make it difficult to change the flow of large-scale control structures, where the blocks that these control are unchanged. As the controlled blocks can be arbitrarily large this may be very costly in terms of reanalysis. However, the locality information can indicate which pieces of text corresponding to well-formed syntactic entities are unchanged, and thus only the surrounding context will require reanalysis.

The case of large-scale change is perhaps less obvious and is more important, so we consider an example program fragment:

```
while ( x > 0 ) {
        y = f(x);
        /* lots more statements */
            ......
        /* finished all the hard work now */
        x = x-1;
       } /* end while */
```

The programmer then realises that the variable `x` can only have the values 0 or 1 when the while loop is encountered, and amends the program by changing the `while` statement to an `if` and deleting the line `x = x-1`:

```
if ( x > 0 ) {
        y = f(x);
        /* lots more statements */
            ......
        /* finished all the hard work now */
       } /* end if */
```

By examining the locality of change the parser/compiler can realise that the sub-expression `x > 0` and the main group of statements have been unchanged, and thus retains the old syntax tree for them, just reparsing the token `if` and the statement brackets "{}".

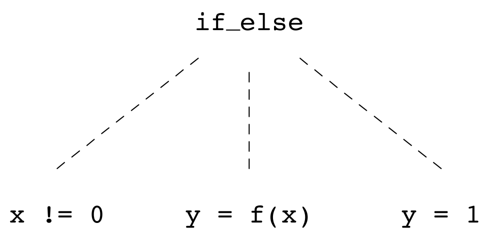
Where the grammar is ambiguous this approach may yield the non-preferred interpretation and thus the syntax tree may need slight jiggering. Alternatively, even though the sub-expression is still the same, it may be the case that there is no proper parsing based on the assumption of the sub-tree being the correct parsing for its text fragment; this would force one to reanalyse more (unchanged) text. Again, it depends on the grammar whether this is likely or even possible. In the second case, there may be no correct parsing because of a syntax error. Working on the assumption that the previously parsable section is correct, one may be able to make far better error reports (and perhaps intelligent suggestions) than normal. In either case, even where there is no error, the change in meaning of the sub-expression may be intentional, but there is a good chance that it is accidental and warnings could be given. For example, consider the "dangling else" problem, a classic example where the syntax tree might need jiggering. The programmer starts off with the following fragment:

```
if ( x != 0 )
        y = f(x);
else
        y = 1;
```

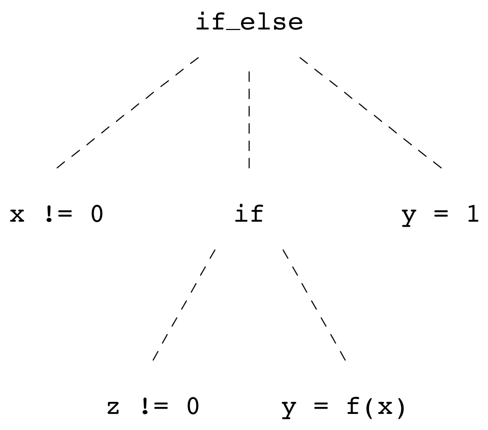
Then, realising that `f(x)` causes an overflow if the global variable `z` is zero, the programmer amends it to leave `y` unchanged in this case:

```
if ( x != 0 )
    if ( z != 0 )
        y = f(x);
else
        y = 1;
```

The original parse tree would have the following form:



Only the text corresponding to the `then` arm would have been altered, so the amended tree would be:



However, the analyser would realise that this was taking the wrong rule for disambiguating the dangling `else` and it should be bound to the innermost `if`. The tree would finally read:

```
                if
              /    \
             /      \
            /        \
           /          \
          /            \

     x != 0       if_else
                 /    |    \
               /      |      \
             /        |        \
           /          |          \
         /            |            \

     z != 0      y = f(x)      y = 1
```

At this stage it would be worth giving a warning, as although this change in the form of the syntax tree might have been just what was intended (and a good reason for using text editing rather than syntactic operations), it could easily have been a mistake. In this example, the reparsing was of course wrong with respect to the programmer's intentions and the warning would have been appreciated.

8.6 Discussion

We have seen in this chapter how dynamic pointers can be given a clean semantic definition in terms of pointer spaces. The pull function that is associated with any operation provides a natural and consistent way to update the state of an editor or application. In order to understand the mapping between display and application and layered design in general, we considered relations between pointer spaces called projections, and the composition of these.

The translation of block pointers was found to be quite complex and this is evidenced by the various inconsistent ways atomic and block pointers are updated in existing systems. Using dynamic pointers to discuss these issues outside of a particular system helps us to form an understanding of the issues, even if no single answer can be found.

We have discussed how some of the more detailed analysis of dynamic pointers using *sub-object projections* and *locality* information about projections and operations can help us to express interface properties and can be used for system optimisations.

Finally, we considered many examples of how dynamic pointers are and could be used. Clearly, one could go on thinking of such uses. Some of those suggested can be implemented now under existing environments, but many rely on the pointers being an integral part of the environment. I believe that advanced interactive environments will contain entities similar to dynamic pointers, and if they are not recognised as a single unifying concept they will instead be implemented in diverse incompatible ways, yielding a clumsy and unpredictable interface.