Appendix I

Notation

Specific notations are introduced in the course of the book for different purposes. These are usually of local significance, and it would not help to list them out of context. The notations used for general concepts, sets, sequences, tuples and functions are given below. There is also a short description of the notation for structures used in Chapter 8.

I.1 Sets

Standard mathematical notation is used for these: {...} to enclose enumerated sets, $\cup$ and $\cap$ for set union and intersection, $\times$ for Cartesian product and $\in$ and $\subset$ for membership and subset. The following notation applies:

$\mathbb{P}X$ – the power set of X, that is, the set of all subsets of X

$\mathbb{F}X$ – the set of finite subsets of X

$\| X \|$ – the cardinality (number of elements in) X

$\mathbb{N}$ – the set of natural numbers, {0,1,2,...}

I.2 Sequences

Many of the models deal with sequences, or lists of values. These are usually finite but of arbitrary length. The following notation is used for sequences:

X^* – the set of all sequences of elements from the set X

X^+ – the set of all non-empty sequences of elements from the set X

null – the empty sequence

[...] – encloses a sequence, so that [a, b, c] is the sequence with a as its first element, b as its second and c as its third and last, [] on its own also represents an empty sequence

(...) – used on occasions to enclose sequences: this is more normal in mathematics, and less obtrusive where there is no confusion

"..." – quotes enclose character sequences

The sequences are glued together using three operations:

: – adds an element to the front of a sequence, e.g.:

$$x : [\ a, b, c\ ] \quad = \quad [\ x, a, b, c\ ]$$

:: – adds an element to the end of a sequence, e.g.:

$$[\ a, b, c\ ] :: x \quad = \quad [\ a, b, c, x\ ]$$

; – concatenates sequences, e.g.:

$$[\ a, b, c\ ] ; [\ x, y, z\ ] \quad = \quad [\ a, b, c, x, y, z\ ]$$

Where the meaning is clear juxtaposition is sometimes used to mean any of the above three, so that if a is an element and p and q are sequences:

$$\begin{array}{lcl} a\ p & = & a : p \\ p\ a & = & p :: a \\ p\ q & = & p\ ;\ q \end{array}$$

The more specific notation is used where there is likely to be confusion lexically (because of the complexity of expressions) or semantically, in particular, when dealing with sequences of sequences.

Sequences are intrinsically ordered:

$\leq$ – denotes *initial* subsequence, so that:

"ab" $\leq$ "abc"
but **not** "bc" $\leq$ "abc"

This same symbol is also used with its normal meaning for numbers and for information ordering on lattices and on view spaces (Chapter 9).

I.3 Tuples

Round brackets are often used to enclose tuples, leaving it to context to disambiguate them from sequences. Angle brackets $< \ldots >$ are also used. In particular, they are always used for major tuples denoting complete models. These occur in free text and the angle brackets emphasise the formal nature of the contents, in contrast to textual parentheses.

I.4 Functions

The bracketing notation $f(x)$ is used for function application rather than lambda calculus juxtaposition. The only exception to this is the translation Tf of f in Chapter 9, which follows the notation of Bancilhon and Spyratos (1981) whose work it extends. In addition, note the following:

$f : X \to Y$
: – f is a function from the set X to the set Y; if f is partial its domain is defined close to its definition, otherwise it should be assumed to be total

dom f
: – the domain of f, that is, the set of elements for which f is defined

range f
: – the range of f, that is, the set of elements that are possible values of $f(x)$

id_X – the identity function from the set X to itself

$f|_X$ – the function f restricted to X, that is:

$$f|_X(x) = \begin{array}{ll} f(x) & \textbf{if } x \in X \\ \textit{undefined} & \textbf{otherwise} \end{array}$$

I.5 Structures – functors

Chapter 8 deals with general parameter structures. For example, if X is a set of interest we might want to talk about the following functions in a uniform manner:

$$a : X \times \mathbb{N} \times X \to X$$
$$b : Char \times X \to X$$

We will use the notation that both are of type $F[X] \to X$ where $F[X]$ represents the structure with elements from X. So for a and b the structures are:

$$F_a[X] = X \times \mathbb{N} \times X$$
$$F_b[X] = Char \times X$$

The structure $F[..]$ can be applied to any set, so that if Y is a set:

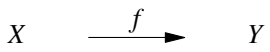
$$F_a[Y] = Y \times \mathbb{N} \times Y$$

Given any function $f: X \to Y$ we can generalise this to a function from $F[X]$ to $F[Y]$, mapping the corresponding elements of the structure. To be strict perhaps one ought to write $F[f]$ for this function, but the meaning is always

obvious from context. In category theory $F[\ldots]$ would be called a functor. This description is partly repeated when it is first used in Chapter 8 to remind the reader.

I.6 Reading function diagrams

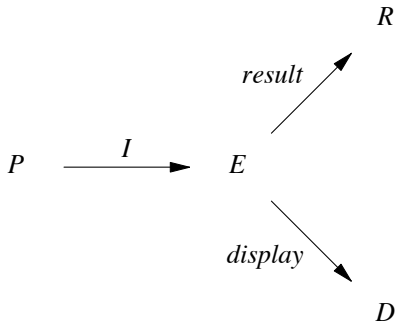
In many places diagrams are used to show the interaction of various functions. In these diagrams, sets are the nodes and arrows the functions between them. The simplest such diagram has only two nodes and one arrow, e.g:



This simply says that f is a function from the set X to the set Y. That is:

$$f : X \rightarrow Y$$

A more complex example is found in Chapter 3:



This says that there are four sets, P, E, R, and D. I is a function from P to E, *result* is a function from E to R and *display* is a function from E to D. In other words:

$$\begin{array}{lcl} I: & P & \rightarrow E \\ result: & E & \rightarrow R \\ display: & E & \rightarrow D \end{array}$$

Neither of these diagrams has more then one path from place to place. However, this need not be the case. A special sort of function diagram is a commuting diagram. In these diagrams there is more than one way to get from place to place,

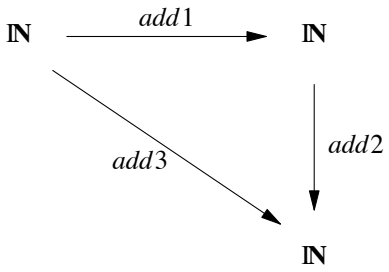
but it "doesn't matter" which path you take.

Consider the natural numbers $\mathbb{N}$ and the three functions $add1$, $add2$ and $add3$, which add 1, 2 or 3 to their arguments, respectively. For example, $add2$ is defined as:

$$add2 : \mathbb{N} \rightarrow \mathbb{N}$$

$$add2(n) = n + 2$$

We can draw the following diagram:



This diagram commutes because it doesn't matter whether we first add 1 and then add 2 (i.e. left and then down) or whether we just add 3 (take the diagonal path). In terms of the functions:

$$\forall n \in \mathbb{N} \quad add2\,(\,add1(n)\,) = add3\,(n)$$

Typically, the sets are different and the diagrams may have many more paths through them.

Appendix II

A specification of a simple editor using dynamic pointers

Pointer spaces can be used to produce an extensible specification for a layered display editor. The display editor consists of three layers:

- *Display* – The actual display as seen by the user. This includes a representation of the cursor, and commands at this level include mouse selection.
- *Text* – A textual representation of the objects of interest, bridging the gap between the objects and the bounded display.
- *Simple strings* – The underlying objects being edited.

In this simple design, the extra text level might be seen as a trifle excessive, but even here it helps to factor the design. This layering becomes essential in more complex systems, where the mapping between the application objects and their textual representation is less straightforward. Further, the layering is useful in proving user interface properties. The text representation is what the user is likely to obtain as an observable effect, and thus a proof of observability can be factored into a proof first that the whole of the text can indeed be observed, and second that the objects can be obtained from some parsing of the text.

We will address the design in several stages. First the three pointer spaces, and the projections between them will be introduced. Then in §II.2 we will see how these can be formed into an editor state. In the next section we will see how the design is extensible by adding a find/replace operation. Finally, we will summarise the insight gained from this example.

II.1 The pointer spaces and projections

In this section we describe the pointer spaces and projections necessary for the three layers of the editor design. At the surface is a simple grid of characters, representing the user's screen. This display is assumed to also have a single cursor *between* characters. This would be presented on most terminals as a

cursor over a character, and the last column would have to be left blank in order to represent the last cursor position:

$$Disp = Array\ [\ height, width\ ]\ of\ Char$$
$$P_{disp} = \{\ 1, \ldots, height\ \} \times \{\ 0, \ldots, width\ \}$$
$$\forall\ disp \in Disp \qquad vptrs(\ disp\ ) = P_{disp}$$

The intermediate level is two-dimensional text, a ragged-edged sequence of lines of printable characters:

$$Text = list\ of\ list\ of\ Char - \{\ [\ ]\ \}$$
$$P_{text} = \mathbb{N} \times \mathbb{N}$$
$$(\ n, m\ ) \in vptrs(\ text\ ) \quad \Leftrightarrow \quad n \in \{\ 1, \ldots, length(\ text\ )\ \}$$
$$\textbf{and}\ \ m \in \{\ 0, \ldots, length(\ text[\ n\ ]\ )\ \}$$

Note that the empty text is represented by "[[]]" (the sequence consisting of a single empty line), and in general the last line of a text is *not* assumed to have an implicit trailing new-line. The disallowing of "[]" (the empty sequence) as a text is thus quite consistent.

The projection between text and display is simple framing, and requires only a single text pointer as parameter specifying the offset of the "window" into the text:

$$Disp\_struct[\ P_{text}\ ] = P_{text}$$

$$proj_{disp} :\ P_{text} \times Text \rightarrow Disp \times (\ P_{text} \rightarrow P_{disp}\ ) \times (\ P_{disp} \rightarrow P_{text}\ )$$
$$proj_{disp}(\ (\ n, m\ ), text\ ) = disp, fwd, back$$

The display *disp* is constructed as the relevent section of *text*, with the locations that don't have corresponding *text* locations filled with blanks:

$$\begin{aligned} disp[\ i, j\ ] \quad &= \quad text[\ i + n - 1\ ][\ j + m\ ] \\ &\qquad\qquad \textbf{if}\ \ i + n - 1 \le length(\ text\ ) \\ &\qquad\qquad \textbf{and}\ \ j + m \le length(\ text[\ i + n - 1\ ]\ ) \\ &= \quad space \qquad \textbf{otherwise} \end{aligned}$$

The *fwd* map normally merely subtracts the window offset $(\ n, m\ )$, but has to return extremal values when the text pointer is outside the window. Note that for this and the *back* map, the last clause is the "normal" case after all exceptions have been dealt with:

$$
\begin{array}{l}
fwd(\,(\,n', m'\,)\,) \;=\; i, j \\
\textbf{where} \\
\qquad \textbf{if}\ n' < n \quad \textbf{then}\ \ (\,i, j\,) \;=\; (\,1, 0\,) \\
\qquad \textbf{else if}\ n' \geq n + height \quad \textbf{then}\ \ (\,i, j\,) \;=\; (\,height, width\,) \\
\qquad \textbf{else if}\ m' < m \quad \textbf{then}\ \ (\,i, j\,) \;=\; (\,n' - n + 1, 0\,) \\
\qquad \textbf{else if}\ m' > m + width \quad \textbf{then}\ \ (\,i, j\,) \;=\; (\,n' - n + 1, width\,) \\
\qquad \textbf{else}\ \ (\,i, j\,) \;=\; (\,n' - n + 1, m' - m\,)
\end{array}
$$

The *back* map has similar problems with "out of range" display pointers, this time ones which represent blank screen locations:

$$
\begin{array}{l}
back(\,(\,i, j\,)\,) \;=\; n', m' \\
\textbf{where} \\
\qquad \textbf{if}\ n + i > length(\,text\,) \\
\qquad \textbf{then}\ \ (\,n', m'\,) \;=\; (\,length(\,text\,), length(\,text[\,length(\,text\,)\,]\,)\,) \\
\qquad \textbf{else if}\ m + j > length(\,text[\,n + i\,]\,) \\
\qquad \textbf{then}\ \ (\,n', m'\,) \;=\; (\,n + i, length(\,text[\,n + i\,]\,)\,) \\
\qquad \textbf{else}\ \ (\,n', m'\,) \;=\; (\,n + i, m + j\,)
\end{array}
$$

The text, in turn, is represented by a projection from a single-dimensional string of characters with embedded new-lines:

$$
\begin{array}{l}
String \;=\; list\ of\,(\,Char + \{nl\}\,) \\
P_{string} \;=\; \mathbb{N}
\end{array}
$$

$$
vptrs(\,string\,) \;=\; \{\,0, \ldots, length(\,string\,)\,\}
$$

This has various operations defined on it, character insertion and deletion, and incrementing and decrementing of pointers:

$$
insert\colon\ P_{string} \times Char \times String \;\rightarrow\; String \times (\,P_{string} \rightarrow P_{string}\,)
$$

$$
\begin{array}{l}
insert(\,n, c, string\,) \;=\; string', pull \\
\textbf{where} \\
\qquad string' \;=\; string[\,1, \ldots, n\,] :: [\,c\,] \\
\qquad\qquad\qquad\qquad :: string[\,n + 1, \ldots, length(\,string\,)\,] \\
\qquad pull(\,n'\,) \;=\; n' \qquad \textbf{if}\ n' < n \\
\qquad\qquad\quad\ = \; n' + 1 \quad \textbf{if}\ n' \geq n
\end{array}
$$

$$
delete\colon\ P_{string} \times String \;\rightarrow\; String \times (\,P_{string} \rightarrow P_{string}\,)
$$

$$
\begin{array}{l}
delete(\,n, string\,) \;=\; string', pull \\
\textbf{where} \\
\qquad string' \;=\; string[\,1, \ldots, n - 1\,] :: string[\,n + 1, \ldots, length(\,string\,)\,] \\
\qquad pull(\,n'\,) \;=\; n' \qquad \textbf{if}\ n' < n \\
\qquad\qquad\quad\ = \; n' - 1 \quad \textbf{if}\ n' \geq n
\end{array}
$$

$$succ\colon\ P_{string} \times String \rightarrow P_{string}$$
$$pred\colon\ P_{string} \times String \rightarrow P_{string}$$

$$succ(\ n, string\ ) = n+1 \quad \textbf{if}\ \ n < length(\ string\ )$$
$$= n \quad \textbf{otherwise}$$
$$pred(\ n, string\ ) = n-1 \quad \textbf{if}\ \ n > 0$$
$$= n \quad \textbf{otherwise}$$

Note that the successor and predecessor functions both require the string as an argument, so that they can do range checking. In fact, we can see that the predecessor function could do without; however, it is better always to require an object context for pointer operations, as it leaves room for alternative representations – we could have had pointers representing distance back from the end of the string!

We can now construct the obvious projection from strings to texts:

$$Proj\_struct\ \text{ is empty}$$

$$proj\colon\ String \rightarrow Text \times (\ P_{string} \rightarrow P_{text}\ ) \rightarrow (\ P_{text} \rightarrow P_{string}\ )$$

$$proj(\ s\ ) = text, fwd, back$$

The maps *fwd* and *back* and the result *text* are defined using two arrays of string pointers, *line_starts* and *line_ends*, both consisting of valid pointers for *s*:

$$line\_starts\colon\ \ list\ of\ vptrs(\ s\ )$$
$$line\_ends\colon\ \ list\ of\ vptrs(\ s\ )$$

The former is taken to have a pointer to the beginning of each line in the string, and the latter to the end of the line. Thus $length(\ line\_starts\ )$ is the number of lines in the string and $line\_starts[\ n\ ]$ is a pointer to the beginning of the *n*th line. We can then define *text* by the following properties:

$$length(\ text\ ) = length(\ line\_starts\ ) = length(\ line\_ends\ )$$
$$text[\ i\ ] = s[\ line\_starts[\ i\ ]+1, \ldots, line\_ends[\ i\ ]\ ]$$

and we obtain *fwd* and *back* fairly directly from *line_starts* and *line_ends*:

$$fwd(\ p\ ) = (\ n, m\ )$$
$$\textbf{st}$$
$$line\_starts[\ n\ ] \leq p \leq line\_ends[\ n\ ]$$
$$m = p - line\_starts[\ n\ ]$$
$$back(\ (\ n, m\ )\ ) = line\_starts[\ n\ ] + m$$

Finally, *line_starts* and *line_ends* themselves are defined by:

$$
\begin{array}{l}
\textbf{let} \quad len \;=\; length(\, line\_starts \,) \\
line\_starts[\, 1 \,] \;=\; 0 \\
line\_ends[\, len \,] \;=\; length(\, s \,) \\
\forall\; i \in \{\, 1, \ldots, len \,\} \\
\qquad nl \notin s[\, line\_starts[\, i \,] + 1, \ldots, line\_ends[\, i \,] \,] \\
\forall\; i < len \\
\qquad line\_starts[\, i + 1 \,] \;=\; line\_ends[\, i \,] + 1 \\
\qquad \textbf{and} \;\; s[\, line\_ends[\, i \,] + 1 \,] \;=\; nl
\end{array}
$$

The definition given is not constructive; however, one can easily produce an effective procedure for calculating *text*, *line_starts* and *line_ends* from a string.

II.2 Construction as state model

We can now use the preceding pointer spaces and projections to define an editor state. Again we will do this in two stages, first defining an editor for texts, with no finite display, then adding state to define the display.

The text editor state simply consists of a string and a string pointer to act as an insertion point:

$$E_{text} \;=\; String \times P_{string}$$

Its interface is a text and a text pointer (the insertion point), yielded by application of the projection:

$$
\begin{array}{l}
display_{text} :\; E_{text} \;\rightarrow\; Text \times P_{text} \\
display_{text}(\, (\, s,\, p \,) \,) \;=\; text, tptr \\
\textbf{where} \\
\qquad text, fwd, back \;=\; proj(\, s \,) \\
\qquad tptr \;=\; fwd(\, p \,)
\end{array}
$$

There are five operations defined. The first two are simple insertion-point moves and the third sets the insertion point to a given text position (for use later with mouse selection). These three have no effect on the pointers, so the *pull* function is just the identity and is omitted from their definitions. The others, insertion and deletion of characters, alter the object and yield *pull* functions so that anything using this can convert its own text pointers. Note that the user of the package has no knowledge of the implementation using strings; the interface is entirely in terms of texts and text pointers:

$\{$ *movement* $\}$:

$move\_right_{text} : \ E_{text} \rightarrow E_{text}$
$move\_left_{text} : \ E_{text} \rightarrow E_{text}$

$move\_right_{text}(\ (\ s, \ p \ ) \ ) \ = \ s, \ succ(\ p, \ s \ )$
$move\_left_{text}(\ (\ s, \ p \ ) \ ) \ = \ s, \ pred(\ p, \ s \ )$

$\{$ *selection* $\}$:

$select_{text} : \ P_{text} \times E_{text} \rightarrow E_{text}$

$select_{text}(\ p_{new}, \ (\ s, \ p_{old} \ ) \ ) \ = \ s, fwd(\ back(\ p_{new} \ ) \ )$

$\{$ *update* $\}$:

$insert_{text} : \ (\ Char + \{ \ nl \ \} \ ) \times E_{text} \rightarrow E_{text} \times (\ P_{text} \rightarrow P_{text} \ )$
$delete_{text} : \ E_{text} \rightarrow E_{text} \times (\ P_{text} \rightarrow P_{text} \ )$

$insert_{text}(\ c, \ (\ s, \ p \ ) \ ) \ = \ (\ s', \ p' \ ), \ pull_{text}$
where
$p' \ = \ pull(\ p \ )$
$s', \ pull \ = \ insert(\ p, \ c, \ s \ )$
$pull_{text} \ = \ fwd' \circ pull \circ back$
$text, fwd, back \ = \ proj(\ s \ )$
$text', fwd', back' \ = \ proj(\ s' \ )$

$delete_{text}(\ (\ s, \ p \ ) \ ) \ = \ (\ s', \ p' \ ), \ pull_{text}$
where
$p' \ = \ pull(\ p \ )$
$s', \ pull \ = \ delete(\ p, \ s \ )$
$pull_{text} \ = \ fwd' \circ pull \circ back$
$text, fwd, back \ = \ proj(\ s \ )$
$text', fwd', back' \ = \ proj(\ s' \ )$

Note the similarity between the definitions of $insert_{text}$ and $delete_{text}$. It is therefore very easy to prove properties of E_{text} and its operations from those of *String* and its operations, for instance, $delete_{text}(\ insert_{text}(\ c, \ e \ ) \ ) = e$.

We construct the full editor by adding the display structure, which is a single text pointer:

$E_{disp} \ = \ E_{text} \times P_{text}$

$\forall \ (\ e_{text}, \ p \ ) \in E_{text} \qquad p \in vptrs(\ text \ )$
where
$text, \ ip \ = \ display_{text}(\ e_{text} \ )$

This has as its interface a display with cursor, obtained by using the display projection on the text obtained from e_{text}:

$$display_{disp} : E_{disp} \rightarrow Disp \times P_{disp}$$

$$display_{disp}((e_{text}, p)) = disp, ip_{disp}$$
where
$$ip_{disp} = fwd(ip_{text})$$
$$disp, fwd, back = proj_{disp}(p, text)$$
$$text, ip_{text} = display_{text}(e_{text})$$

As an observability condition, we would always like the text insertion point to be on screen:

$$\forall (e_{text,} p) \in E_{disp} \qquad ip_{text} \in back (P_{disp})$$

We recognise this as a *static invariant*, relating the display and state at an instant. In order to maintain this, when the normal course of operations breaks this invariant, we need an internal adjustment function to be used after each operation:

$$adjust : E_{text} \times P_{text} \rightarrow P_{text}$$

$$adjust(e, (p\_line, p\_col)) = (p\_line', p\_col')$$
where
$$\textbf{if } ip\_line - p\_line + 1 \in \{ 1, \ldots, height \} \textbf{ then } p\_line' = p\_line$$
$$\textbf{else } p\_line' = \max (1, ip\_line - height/2)$$
$$\textbf{if } ip\_col - p\_col + 1 \in \{ 1, \ldots, width \} \textbf{ then } p\_col' = p\_col$$
$$\textbf{else } p\_col' = \max (1, ip\_col - width/2)$$
$$text, (ip\_line, ip\_col) = display_{text}(e)$$

We can now use this to define the final display editor operations. These are simply the operations on E_{text}, followed by *adjust*:

{ *movement* }:
$$move\_right_{disp} : E_{disp} \rightarrow E_{disp}$$
$$move\_left_{disp} : E_{disp} \rightarrow E_{disp}$$

$$move\_right_{disp}((e_{text}, p)) = (e'_{text}, p')$$
where
$$p' = adjust(e'_{text}, p)$$
$$e'_{text} = move\_right_{text}(e_{text})$$

$move\_left_{disp}(\ (\ e_{text},\ p\ )\ )\ =\ (\ e'_{text},\ p'\ )$
where
$p'\ =\ adjust(\ e'_{text},\ p\ )$
$e'_{text}\ =\ move\_left_{text}(\ e_{text}\ )$

{ *selection* }:
$select_{disp}\ :\ P_{disp}\ \times\ E_{disp}\ \rightarrow\ E_{disp}$

$select_{disp}(\ p_{new},(\ e_{text},\ p\ )\ )\ =\ (\ e'_{text},\ p\ )$
where
$e'_{text}\ =\ e,\ back(\ p_{new}\ )$
$e_{text}\ =\ e,\ p_{old}$
$back\ =\ proj_{disp}(\ p,\ text\ ).\ back$
$text\ =\ proj_{text}(\ e\ ).\ obj$

N.B. No adjustment is necessary, as $back(\ p_{new}\ )$ is in $back(\ P_{disp}\ )$ by definition.

{ *update* }:
$insert_{disp}:\ (\ Char\ +\ \{\ nl\ \}\ )\ \times\ E_{disp}\ \rightarrow\ E_{disp}$
$delete_{disp}:\ E_{disp}\ \rightarrow\ E_{disp}$

$insert_{disp}(\ c,(\ e_{text},\ p\ )\ )\ =\ (\ e'_{text},\ p'\ )$
where
$p'\ =\ adjust(\ e'_{text},\ pull(\ p\ )\ )$
$e'_{text},\ pull\ =\ insert_{text}(\ c,\ e_{text}\ )$

$delete_{disp}(\ (\ e_{text},\ p\ )\ )\ =\ (\ e'_{text},\ p'\ )$
where
$p'\ =\ adjust(\ e'_{text},\ pull(\ p\ )\ )$
$e'_{text},\ pull\ =\ delete_{text}(\ e_{text}\ )$

II.3 Adding features

As well as dividing concerns very clearly between the various levels, the design is very easily extensible. For instance, imagine we designed an additional operation *find_replace* for strings:

$$find\_replace:\ String\ \times\ String\ \times\ String\ \rightarrow\ String\ \times\ (\ P_{string}\ \rightarrow\ P_{string}\ )$$

where $find\_replace(\ f,\ r,\ s\ )$ replaces all occurrences of f in s and puts r instead. We can trivially extend the rest of the editor to include this:

$$find\_replace_{text} : String \times String \times E_{text} \rightarrow E_{text} \times (P_{text} \rightarrow P_{text})$$

$$find\_replace_{text}(f, r, (s, p)) = (s', p')$$

where

$$p' = pull(p)$$
$$s', pull = find\_replace(f, r, s)$$
$$pull_{text} = fwd' \circ pull \circ back$$
$$text, fwd, back = proj(s)$$
$$text', fwd', back' = proj(s')$$

$$find\_replace_{disp} : String \times String \times E_{disp} \rightarrow E_{disp}$$

$$find\_replace_{disp}(f, r, (e_{text}, p)) = (e'_{text}, p_{text})$$

where

$$p' = adjust(e'_{text}, pull(p))$$
$$e'_{text}, pull = find\_replace_{text}(f, r, e_{text})$$

The final lexical level of the editor has not been included here. Some commands are simple: insert, delete and movement would map onto single keystrokes and selection would be a mouse click. The find/replace operation would require extra mechanisms enabling the user to manipulate the search strings. Of course, we already have an editor for working on strings (in fact we have two!), and this could be used. We could add such a search string editor as a special feature, or better it could be included in an environment that manages objects and object–object operations like find/replace.

II.4 Discussion

We've seen how a simple editor can be built up out of simple specification components using pointer spaces. Translation of mouse coordinates inwards, and cursor positions on display are achieved using the standard *back* and *fwd* maps. We've also seen how easy it is to add facilities such as find/replace. Not only was this easy, but the mechanism is identical for any operation defined on the underlying data type that can supply a *pull* function. This is very important, as it means that find/replace can be thought of as a *tool* used by the editor rather than an integral part of the editor. Other filters could be included dynamically in the editor's repertoire, for instance an intelligent spelling checker or a program pretty printer. This extends significantly the power of environments such as Unix which make heavy use of simple filters as tools.