BR[illegible] 6

AD-A204 834

RSRE

MEMORANDUM No. 4247

ROYAL SIGNALS & RADAR ESTABLISHMENT

A Z SPECIFICATION
OF THE MaCHO INTERFACE EDITOR

Author: A W Wood

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

89 2 27 200

Royal Signals and Radar Establishment
Memorandum Number 4247

Title:	A Z Specification of the MaCHO Interface Editor
Author:	A.W.Wood
Date:	November 1988

Abstract

This document describes the basic editor part of the user interface for the SMITE secure computer architecture using the mathematical notation known as Z. Operations that are available to the user, and their effects on the screen display, are specified in conjunction with descriptions of the auxiliary functions and data structures necessary to support them. This specification will be used to implement a powerful yet trustworthy interface for the initiation and control of security related transactions.

[illegible]

Introduction

The MaCHO (Mouse and Cartouches for Handling Objects) Interface, or MaCHO-I for short, is designed to be the primary application independent user interface for the SMITE secure capability computer [Terry88, Wiseman88a, Wiseman88b, Wiseman88c]. This document formally describes the salient features of MaCHO-I in the notation of Z, a specification language based on the mathematics of typed set theory developed by the Programming Research Group at Oxford University. The reader who is unfamiliar with Z is advised to read, or at least have available for consultation, the many good reference manuals available on the language [Sufrin86, Spivey88] before embarking on the rest of this paper.

MaCHO-I is strongly based on the interface of the Flex computer [Foster82], in particular in its use of the graphical block (henceforth referred to as the gblock) as a fundamental data object. As its name suggests, this data structure is one that is directly representable graphically on the display screen and which is composed of a set of objects grouped together to form a whole. Gblocks can thus take on many forms of varying sizes but, more importantly, they can contain other gblocks in their object set. For instance, if we view a line of characters as a gblock then several lines may be combined to form a gblock called a paragraph; similarly, several paragraphs may make a page. (For more on the uses of gblocks see [Core87].)

Whilst both the MaCHO and Flex interfaces are founded upon gblocks, it is important to note that the former, MaCHO-I, can neither manipulate them to such a great extent as the latter, nor does it even recognise the full spectrum of types of gblock that Flex does (approximately two thirds are not present). However, despite this reduced functionality, the MaCHO Interface is being constructed to be superior to its Flex counterpart in possessing those properties that are of overriding concern in the environment of the SMITE project - complete trustworthiness and total correctness.

To describe the design of MaCHO-I in a clearly comprehensible manner, this document has been split into a number of discrete sections. Section 1 is concerned with an informal description of gblocks and other data structures as they are used in MaCHO-I. Sections 2 and 3 formalise, in Z, the notions of Section 1 and also introduce some operators to process them. Section 4 defines those functions which are available to the user whose effects on the various data structures of the interface are specified in sections 7 through to 20. Section 5 deals with those aspects of the interface associated with defining views of the data by windows and section 6 defines those simple schemas which are not constrained to be used in conjunction with any one particular function but rather with many spread through the text.

Contents

Section 1: Description of MaCHO-I Data Structures

The display of the MaCHO Interface (i.e. what the user sees on the screen) is, at first glance, very simple - to the user it appears as a (mostly) blank space over which they can reposition and type text at will. To select a point on the screen for the cursor requires a simple movement of the mouse over a tablet.

Of course, the real situation is much more complicated. Firstly, the display is composed of many non-overlapping gblocks that can be selected by the mouse, usually together with several void portions of the screen, that is, areas which are not covered by any part of a gblock and can NOT be selected by the mouse. Secondly, the position of the mouse on the tablet does not necessarily bear any relation to the displayed cursor - the system which reads and interprets the location of the mouse is entirely seperate to that which displays the cursor. This means that one of either the cursor or the mouse can be moved without affecting the other. At all times though, the absolute position of the mouse on the tablet is conveyed to the user via a small arrow, known as a pointer, drawn on the screen.

In MaCHO-I there are four distinct types of gblock:

- the line
- the vervec
- the horvec
- the cartouche

We will examine each of these types of gblock in turn and describe their representation in the explanatory diagrams to be used throughout the remainder of this text.

Line

A line is a simple sequence of characters, as may be expected. The gblock containing this sequence will be represented as the string enclosed by double quotes e.g. "Hello Everybody".

Vervec

A vervec is a vertical vector of gblocks and so, for instance, one line atop another constitutes a vervec. Vervecs notionally run from the top of the page or screen to the bottom and so in all diagrams, a vervec will be shown as a series of rectangles with this orientation:

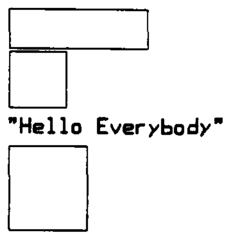


Figure 1: An Example Vervec

Figure 1 shows a vervec whose third element is a line type gblock containing the string "Hello Everybody"; the contents of the other elements in the vervec are not specified. Note that the elements of the vervec align at their leftmost edge.

Horvec:

A horvec is a horizontal vector of gblocks which notionally runs from left to right on the screen. For instance, three vervecs alongside each other would constitute a horvec. In a fashion similar to that for vervecs, horvecs are drawn as a sequence of rectangles in the appropriate orientation:

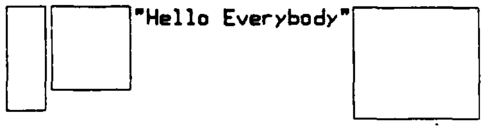


Figure 2: An Example Horvec

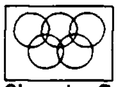
Again, we notice an important alignment property: the gblocks of a horvec align at their topmost points.

(The reader should note that the rectangles in the diagrams for horvecs and vervecs are for the purposes of illustration only - they do not appear in the actual display of MaCHO-I.)

Cartouche

Of the four types of gblock in MaCHO-I , the cartouche is by far the most difficult structure to describe in simple terms. Perhaps surprisingly, however, we can gain some sort of understanding from Chambers 20th Century Dictionary which describes a cartouche as "an ornamental form for receiving an inscription": in MaCHO-I we view the cartouche as a visible representation of a capability (in the computing sense of [Fabry74]) which can take the form of a picture or just simple text.

In its (less common) pictorial form, a cartouche is akin to the well known notion of an icon on such machines as the Apple Macintosh; in its textual form it resembles a *file* name. Whatever form they take though, cartouches in MaCHO-I are always displayed as being enclosed by a rectangular border so that they can be easily distinguished from ordinary text or pictures. For instance, centre_text :Module is a cartouche representing the capability to a file containing the (compiled) source code of a program to centre text, whereas

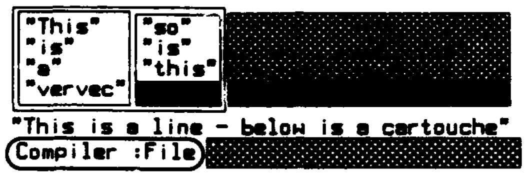


may be a user defined cartouche of the capability to a file holding Olympic Games records. To actually access the underlying files (through their capabilities), certain operations can be directly applied to the cartouches themselves. In subsequent diagrams, we will represent cartouches as 'lozenges', possibly containing text within:

centre_text :Module

Before ending this description of cartouches, it should be pointed out that in the following specification we make no mention of their special properties - we regard them only as another type of gblock to be handled by the interface. Such treatment is sufficient for the purposes of this document.

Although appearing very basic, the above four types of gblock can be combined to form remarkably complex structures:



```
"This"   "so"
"is"     "is"
"a"      "this"
"vervec"
"This is a line - below is a cartouche"
(Compiler :File)
```

Figure 3: A typical display in MaCHO-I

Figure 3 (above) shows a three element vervec. The first gblock in this vervec is of horvec type, the second is a line and the third is a cartouche. In turn, the horvec is composed of two vervecs, both of which are made entirely from line type gblocks. Those parts of the diagram that are shown in black represent void portions of the screen inasmuch as any attempt to point to these regions (using the mouse) will be disallowed. Regions of this nature arise due to the disparity of the sizes of the gblocks in the display. In this example, the horvec contains such a void region due to the fact that one of its elementary gblocks is four lines 'long' and the other is only three. Those regions that are shown 'cross-hatched' are areas of the screen which, although not visibly covered by any objects, are considered to be full of space characters and so they can be selected by the mouse. The effect pointing to these regions depends on the gblock that is on their immediate left - see section 8.

There are three further features of MaCHO-I, each of which plays an important part in creating and maintaining the useability of the interface:

- a coordinate system
- a variable sized cursor
- a stack of remembered elements.

Coordinate System

The display screen has a coordinate system such that the origin (0,0) is at the top left hand corner. From this point, coordinates in the x direction increase to the right; those in the y direction increase downwards, towards the bottom of the screen. Thus all valid screen positions are represented by a pair of positive numbers. We use this coordinate system to determine which gblock has been selected by pointing operations and where gblocks are displayed. The outermost gblock, which represents the entire screen, obviously has its top left corner at (0,0).

The Cursor

In the MaCHO Interface, the size of the cursor shrinks and grows dynamically according to the requirements of the user (but it cannot become larger than the screen nor can it contract to be smaller than a single character). At all times, the objects currently covered by the cursor are displayed in reverse video format and so if the cursor were covering the letter 'a' say, then the user would see a white 'a' against a dark background. In most instances, the cursor does just cover one single character; however, it is possible to group gblocks together to form one large cursor to show for instance several lines in reverse video. It is this grouping mechanism that allows us to delete objects covering significant portions of the screen in one keystroke (cf. "Remembered Elements" below).

Remembered Elements

This is a data structure which holds all the gblocks that the user has deleted from the screen by one method or another during the session. The most recently deleted gblock is at the top of the stack. At any time, this topmost gblock can be reinserted into the display at the place of the current cursor (see "The Cursor" above) and so removed from the remembered elements. Such a scheme gives the user considerable "cut and paste" power since gblocks can be repeatedly put on and taken off the stack until the desired display is achieved. As a method of communicating the state of the remembered elements to the user, a message is shown giving the number of elements on the stack (but not what is actually in it).

This concludes our informal description of the MaCHO Interface and its data structures. Despite the brevity of the preceding discussion, it is hoped that more of the manipulative properties of MaCHO-I will become apparent to the reader from the formal specification to follow.

[GBLOCK]

We introduce the set [GBLOCK] which contains every gblock that can possibly be represented in MaCHO-I. This set is very large, despite the fact that, as stated in Section 1, there are only four distinct types of gblock - we shall see why presently.

The most basic type of gblock is the LINE. It is formally defined as being composed of three subsequences of characters: those before and after the cursor plus those under the cursor itself. The only restriction on this structure is that the cursor sequence is non-empty:

```
┌─LINE ─────────────────────────────
│ before, cursor, after: seq Char
├───────────────────────────────────
│ #cursor ≥ 1
└───────────────────────────────────
```

A CARTOUCHE has only two parts: a gblock and some unspecified quantity taken from the set [VALUE]. (The reader should note that this latter component of a CARTOUCHE is of no concern to this specification - its declaration is given only for the sake of completeness.)

```
┌─CARTOUCHE ──
│ v: VALUE
│ gb: GBLOCK
└─────────────
```

To specify the notions of HORVECs and VERVECs it is necessary to go through two stages. The first consists of a "partial" definition; a second stage presented later expands this to a "total" definition.

A PARTIAL_HORVEC is one made up of three subsequences of gblocks in a manner similar to that of a LINE. The sequences notionally run from left to right.

```
┌─PARTIAL_HORVEC ─────────────────────
│ before, cursor, after: seq GBLOCK
├─────────────────────────────────────
│ #cursor ≥ 1
└─────────────────────────────────────
```

A PARTIAL_VERVEC has an identical specification to that of a PARTIAL_HORVEC but here the elementary gblocks within each subsequence notionally run from top to bottom (see Section 1).

```
PARTIAL_VERVEC ≙ PARTIAL_HORVEC
```

Given these definitions so far, one can easily see that they model the recursive nature of gblocks. It is also noticeable that of the four, only the line type gblock is not recursively defined.

The next stage of the formalisation is to define some method by which we can distinguish between the four types of gblocks or, in other words, given a gblock we can discover its type. In Z, the distinction is modelled as four functions:

```
| line      : GBLOCK ⤀ LINE
| horvec    : GBLOCK ⤀ PARTIAL_HORVEC
| vervec    : GBLOCK ⤀ PARTIAL_VERVEC
| cartouche: GBLOCK ⤀ CARTOUCHE
|------------------------------------------------
| (dom line, dom vervec, dom horvec, dom cartouche)
|                                  partition GBLOCK
```

The predicate informs us that no type of gblock, other than the given four exists and furthermore, that each gblock is of exactly one type.

These four "type determining" functions above allow us to embark on the second stage of the definitions for HORVECS and VERVECS: a "total" horvec is one which contains no horvecs in any of its subsequences and similarly, a "total" vervec contains no vervecs. (We shall dub this the "no X in X" rule).

```
┌─ HORVEC ────────────────────────────────────
│ PARTIAL_HORVEC
├─────────────────────────────────────────────
│ ran(before ⁀ cursor ⁀ after ) ∩ (dom horvec ) = {}
└─────────────────────────────────────────────

┌─ VERVEC ────────────────────────────────────
│ PARTIAL_VERVEC
├─────────────────────────────────────────────
│ ran(before ⁀ cursor ⁀ after ) ∩ (dom vervec ) = {}
└─────────────────────────────────────────────
```

(The reason for the two stages should now be clear - the full definitions require the declaration of the "typing" functions.)

The "no X in X" rule is introduced to exclude confusion for the user - when displayed, there is no visible difference between a vervec containing no vervecs and one containing several but the structural discrepancies can show up when certain functions are performed. As a side effect of its specification, the "no X in X" rule shortens any proofs that we may carry out on the operation of MaCHO-I by eliminating one class of problem from consideration when analysing the subelements of horvecs and vervecs.

One additional constraint is necessary for gblocks, namely that their definitions are non-circular. To realise this notion we make use of a relation contains.

```
| contains: GBLOCK ↔ GBLOCK
|---------------------------------------------------
| ∀a, b: GBLOCK •
|     (a, b) ∈ contains ⇔
|      (a ∈ dom horvec ∧
|       b ∈ ran ((horvec(a)).before ⁀ (horvec(a)).cursor ⁀
|               (horvec(a)).after ))
|      ∨
|      (a ∈ dom vervec ∧
|       b ∈ ran ((vervec(a)).before ⁀ (vervec(a)).cursor ⁀
|               (vervec(a)).after ))
|      ∨
|      (a ∈ dom cartouche ∧ b = (cartouche(a)).gb)
```

contains holds those pairs of gblocks (g,g') such that gblock g' exists somewhere in the internal structure of gblock g. Lines therefore contribute no elements to the domain of contains (but obviously there may be many such gblocks in its range).

One statement now suffices to give us a full embodiment of non-circularity:

$contains^{+} \cap (id\ GBLOCK) = \{\}$

The formal specification of the data structures is now complete but for the sake of brevity and clarity in the remainder of the document, we shall define schemas which allow for their easier manipulation.

For horvecs, vervecs and lines it is useful to be able to refer to only one of their three constituent sequences:

```
[X]
α, β, γ: GBLOCK ⇸ seq X
------------------------------------------------
∀ g: dom horvec •
      α g = h.after ∧ β g = h.before ∧ γ g = h.cursor
   where
      h ≙ horvec( g )
∀ g: dom vervec •
      α g = v.after ∧ β g = v.before ∧ γ g = v.cursor
   where
      v ≙ vervec( g )
∀ g: dom line •
      α g = l.after ∧ β g = l.before ∧ γ g = l.cursor
   where
      l ≙ line( g )
```

It is also useful to be able to bind the three subsequences together:

```
[X]
bind: GBLOCK ⇸ seq X
------------------------------------------------
∀ g: GBLOCK • bind g = ⁀/⟨ β g, γ g, α g ⟩
```

As a sort of inverse to BIND, we need functions which given a sequence of objects (in this case GBLOCKs or CHARs) constructs a single GBLOCK out of them:

```
[X]
make: seq X ⇸ GBLOCK
------------------------------------------------
∀ s: seq₁ X •
   β (make s) = ⟨⟩
   γ (make s) = s
   α (make s) = ⟨⟩
```

Cartouches have different components to lines, horvecs and vervecs and consequently have different methods by which to access them individually:

$$\begin{array}{|l}
\tau : \mathrm{GBLOCK} \twoheadrightarrow \mathrm{GBLOCK} \\
\nu : \mathrm{GBLOCK} \twoheadrightarrow \mathrm{VALUE} \\
\hline
\forall g : \mathrm{dom}\ cartouche \bullet \\
\quad \tau\ g = (cartouche(\ g\ )).gb \\
\quad \nu\ g = (cartouche(\ g\ )).v
\end{array}$$

(The Greek letters used above are chosen to be aide-memoires: α for "after", β for "before" and, in the absence of a Greek 'c', γ for "cursor". τ is derived, somewhat dubiously, from carTouche and ν appears courtesy of its similarity to "v" as in "value"!)

The first basic operator we introduce is one to sum a sequence of natural numbers:

```
│ Σ: seq ℕ → ℕ
├──────────────────────────────
│ Σ ⟨⟩ = 0
│ ∀s: seq₁ ℕ • Σ s = (hd s) + Σ (tl s)
```

To create, alter and move objects in the MaCHO Interface - and indeed any other interface - we must know (or be able to determine) the height and width of the individual objects:

```
═[Y]═══════════════
│ hei, wid: Y → ℕ
└──────────────────
```

Given that we can find the width of single objects, we can now find the width and height (in pixels) of an entire sequence of objects:

```
═[Y]══════════════════════════
│ sumwid, sumhei: seq Y → ℕ
├─────────────────────────────
│ ∀s: seq Y •
│     sumwid s = Σ (s⨾ wid)
│     sumhei s = Σ (s⨾ hei)
└─────────────────────────────
```

and we can also discover the height of the highest and the width of the widest object in a sequence of objects:

```
═[Y]══════════════════════════════
│ maxwid, maxhei: seq₁ Y → ℕ
├─────────────────────────────────
│ ∀ s: seq₁ Y •
│     maxwid s = max (wid⦇ ran s ⦈)
│     maxhei s = max (hei⦇ ran s ⦈)
└─────────────────────────────────
```

For MaCHO-I we also need functions which split sequences according to some general methodology; that is, given a sequence we can return certain parts of it which are determined by the result of a supplied function:

$$
\begin{array}{|l}
\hline
[Y] \\
(\_LOSPLIT\_),\ (\_HISPLIT\_): \\
\qquad\qquad (\mathrm{seq}_1\ Y \times (\mathbb{N} \times (\mathrm{seq}\ Y \rightarrow\!\!\!\!\!\rightarrow \mathbb{N}))) \rightarrow\!\!\!\!\!\rightarrow \mathrm{seq}\ Y \\
\hline
\forall s,\ t: \mathrm{seq}_1\ Y;\ u: \mathrm{seq}\ Y;\ n: \mathbb{N};\ f: \mathrm{seq}\ Y \rightarrow\!\!\!\!\!\rightarrow \mathbb{N} \mid s = t \frown u\ \bullet \\
\qquad f\ t > n \\
\qquad f\ (\mathrm{front}\ t) \le n \\
\qquad s\ LOSPLIT\ (n,\ f) = t \\
\qquad s\ HISPLIT\ (n,\ f) = u \\
\hline
\end{array}
$$

s LOSPLIT (n, f)is that part of s up to, but not including, the first element whose 'value', as determined by function f, is greater than n. s HISPLIT (n, f) returns the other 'half' of s.

A blank gblock is a line type gblock with only a single space character in its cursor:

$$
\begin{array}{|l}
space: Char \\
blank: \mathrm{dom}\ line \\
\hline
\beta\ blank = \langle\rangle \\
\gamma\ blank = \langle\ space\ \rangle \\
\alpha\ blank = \langle\rangle
\end{array}
$$

All of the functions in the MaCHO Interface take as (part of) their input a gblock and a stack of remembered elements, this latter being formally specified as a sequence of gblocks. These two parameters to the function are called the input STATE.

```
┌─STATE──────────────
│ g: GBLOCK
│ stack: seq GBLOCK
└────────────────────
```

Every function delivers another STATE as its output and so in general we have:

```
┌─FUN────────────────
│ f: STATE ⇸ STATE
│
└────────────────────
```

To move the cursor on the display we use:

```
| left, right, up, down: STATE ⇸ STATE
```

To rub out characters to the left of the current cursor (sometimes referred to as backspace) we use:

```
| del_left: STATE ⇸ STATE
```

and to erase those characters "underneath" the current cursor,

```
| del_right: STATE ⇸ STATE
```

Grouping several characters or gblocks together to form a large cursor is performed by

```
| group_left, group_right, group_up, group_down: STATE ⇸ STATE
```

whose inverses are:

```
| ungroup_left, ungroup_right,
| ungroup_up, ungroup_down: STATE ⤔ STATE
```

Having grouped some gblocks together under the cursor, we may wish to delete this structure onto the stack of the input STATE which can be achieved by the function:

```
| del_element: STATE ⇸ STATE
```

Another method of adding elements to the current stack of the input but without altering the display is to duplicate those gblocks which are currently "under" the cursor:

```
| duplicate: STATE ⤔ STATE
```

Once on the current stack of remembered elements, a gblock can be reinserted into the display in two manners - horizontally, that is immediately in front of the cursor, or vertically, i.e. immediately above the cursor:

```
| ins_above, ins_before: STATE ⇸ STATE
```

An entirely new blank line type gblock can be inserted in the display either above or below the current cursor:

| blank_above, blank_below: STATE ⇸ STATE

Whenever the current cursor is covering a cartouche, we can recover the gblock which is associated with the cartouche (= cartouche(cursor).gb) using:

| undo: STATE ⇸ STATE

All of the above functions can be abstracted away into schemas, using schema FUN:

```
Right ≙ [ FUN | f = right ]
Left ≙ [ FUN | f = left ]
Up ≙ [ FUN | f = up ]
Down ≙ [ FUN | f = down ]
DelLeft ≙ [ FUN | f = del_left ]
DelRight ≙ [ FUN | f = del_right ]
GroupLeft ≙ [ FUN | f = group_left ]
GroupRight ≙ [ FUN | f = group_right ]
GroupDown ≙ [ FUN | f = group_down ]
GroupUp ≙ [ FUN | f = group_up ]
UngroupLeft ≙ [ FUN | f = ungroup_left ]
UngroupRight ≙ [ FUN | f = ungroup_right ]
UngroupUp ≙ [ FUN | f = ungroup_up ]
UngroupDown ≙ [ FUN | f = ungroup_down ]
DelElement ≙ [ FUN | f = del_element ]
Duplicate ≙ [ FUN | f = duplicate ]
InsAbove ≙ [ FUN | f = ins_above ]
InsBefore ≙ [ FUN | f = ins_before ]
BlankAbove ≙ [ FUN | f = blank_above ]
BlankBelow ≙ [ FUN | f = blank_below ]
Undo ≙ [ FUN | f = undo ]
```

and their inputs and outputs can be modelled using the schema IO:

IO ≙ [s, s': STATE]

However, two functions implemented in the interface cannot be abstracted as above since their inputs are not just a single STATE: ins_char is the function that is called every time the user hits a key on the keyboard and thus has the struck character as part of its input, and goin is the function that is called when the user moves the cursor between different gblocks on the screen - necessitating the coordinates of the point selected to be included in the input parameters.

| ins_char: (STATE × Char) ⤀ STATE
| goin: (STATE × N × N) → STATE

We can though still model the input and output state parts for ins_char and goin using IO.

Every coordinate in the MaCHO Interface is represented by a pair of non-negative numbers:

$COORD \triangleq (\mathbb{N} \times \mathbb{N})$

A BRECT is a rectangular area defined by the coordinates of its top left and bottom right corners:

```
┌─ BRECT ─────────────────────
│ tl, br: COORD
├─────────────────────────────
│ fst( tl ) ≤ fst( br )
│ snd( tl ) ≤ snd( br )
└─────────────────────────────
```

The windowing scheme of the MaCHO Interface is specified as a function between BRECTs and STATEs where the BRECT defines the window through which the gblock of the STATE it is mapped to is seen. (Recall from Section 1 that the stack of remembered elements in a state is not shown.)

```
┌─ W ─────────────────────────
│ windows: BRECT ⇸ STATE
└─────────────────────────────
```

As may be guessed, however, there are constraints on the elements of the windowing scheme so far presented whose descriptions require the declaration of some auxiliary functions.

Firstly, we define a function which returns the set of coordinates "covered" by a BRECT:

```
│ cover: BRECT → ℙ COORD
├──────────────────────────────────────────────────────
│ ∀b: BRECT •
│    cover( b ) = {x: (fst( b.tl ) .. fst( b.br ));
│                  y: (snd( b.tl ) .. snd( b.br )) • (x, y)}
```

Next, we define a special BRECT which defines the area the user has available to work in and two integers which define the maximum dimensions of any gblock:

```
│ workspace: BRECT
│ xmax, ymax: ℕ
```

Using the above, we expand the specification of the MaCHO windowing scheme by insisting that

- the windows completely cover the available space,
- no two windows overlap,
- and no gblock has a size greater than the maximum allowed.

```
┌─WINDOWS──────────────────────────────────────────────────────
│ W
├──────────────────────────────────────────────────────────────
│ ⋃ cover( dom windows ) = cover( workspace )
│ ∀ p, q: dom windows | p ≠ q • cover( p ) ∩ cover( q ) = {}
│ ∀ s: ran windows • (wid s.g) ≤ xmax ∧ (hei s.g) ≤ ymax
└──────────────────────────────────────────────────────────────
```

Under the constraints of WINDOWS, we can construct an initial system state in which there is only one window whose associated gblock consists of a blank line type gblock and whose stack of remembered elements is empty:

```
┌─INITIAL──────────────────────
│ ΔWINDOWS
├──────────────────────────────
│   windows' = {workspace ↦ s}
│ where
│   │ s: STATE
│   ├──────────────
│   │ s.stack = ⟨⟩
│   │ s.g = blank
└──────────────────────────────
```

Section 6: Basic Schemas

In this section we introduce some basic schemas which will be used in various places throughout the remainder of the document

The first schema is a simple one containing just two natural numbers which can represent an absolute coordinate in the workspace or a relative offset from some point:

```
XY ≙ [ x, y: ℕ ]
ΔXY ≙ [ XY; XY' ]
```

(Although XY is identical to COORD in section 6, we declare XY seperately since it is to be used in a different way.)

Related to the above is a schema which sets the two values in XY to zero:

```
┌─SetO ──────────
│ ΔXY
├────────────────
│ x' = y' = 0
└────────────────
```

Another simple schema to define is one whose action is the null action:

```
┌─Null ──────────
│ IO
│ ΞWINDOWS
├────────────────
│ s' = s
└────────────────
```

The next three schemas define the types of the gblock which are passed in and out of the functions of the interface:

```
┌─VOps ──────────────────
│ IO
├────────────────────────
│ s.g ∈ dom vervec
│ s'.g ∈ dom vervec
└────────────────────────
```

```
┌─LOps ──────────────────
│ IO
├────────────────────────
│ s.g ∈ dom line
│ s'.g ∈ dom line
└────────────────────────
```

```
┌─ HOps ──────────────────┐
│ IO
├─────────────────────────
│ s.g ∈ dom horvec
│ s'.g ∈ dom horvec
└─────────────────────────┘
```

and the next two define the type of the gblock at the top of the stack of remembered elements in a given state:

```
┌─ TopHorvec ──────────────────┐
│ IO
├──────────────────────────────
│ hd (s.stack) ∈ dom horvec
│ s' = s
└──────────────────────────────┘

┌─ TopVervec ──────────────────┐
│ IO
├──────────────────────────────
│ hd (s.stack) ∈ dom vervec
│ s' = s
└──────────────────────────────┘
```

We use XY in a schema which describes the action of "going in" to a single gblock in a sequence of gblocks:

```
┌─ S ──────────────────────────────────┐
│ IO
│ ΞWINDOWS
│ ΞXY
├──────────────────────────────────────
│    #(γ s.g) = 1
│    β s'.g = β s.g
│    γ s'.g = ⟨ result.g ⟩
│    α s'.g = α s.g
│    s'.stack = result.stack
│ where
│    │ st, result: STATE
│    ├───────────────────────────
│    │ st.g = hd (γ s.g)
│    │ st.stack = s.stack
│    │ result = goin( st, x, y )
└──────────────────────────────────────┘
```

S is defined only when the gblock that is entered is the unique element in the cursor sequence of the input state.

The most difficult operation to specify in this section, and probably in the entire document, is that of normalisation so we will go through it in some detail here.

By normalisation, we mean the moulding of horvecs and vervecs into their canonical forms without loss of information. This process has three distinct stages:

- transform the object into a sequence of its constituent gblocks,
- process this sequence to remove any undesirable features,
- partition the processed sequence into three subsequences to form the before, cursor and after sequences of the result.

This first stage is performed by the binding function of section 1: the intermediate stage is responsible for

- correcting any degenerate gblocks,
- removing horvecs from horvecs and vervecs from vervecs, in order to conform to the "no X in X" rule of section 2, and
- merging adjacent line type gblocks (in horvecs only).

For the first of these processes, we need to define what a degenerate gblock is:

```
| degenerate: P GBLOCK
|------------------------------------------------
| degenerate = {g: dom horvec ∪ dom vervec | # (bind g) = 1}
```

- a degenerate gblock is a horvec or a vervec which contains only one gblock. During normalisation of a sequence of gblocks degenerate gblocks are removed and replaced by the single element they contain:

```
| purge: seq GBLOCK → seq GBLOCK
|------------------------------------------------
| purge ⟨⟩ = ⟨⟩
| ∀ g: degenerate • purge ⟨g⟩ = bind g
| ∀ g: GBLOCK\degenerate • purge ⟨g⟩ = ⟨g⟩
| ∀ s, t: seq GBLOCK • purge( s ⁀ t ) = purge( s ) ⁀ purge( t )
```

In a fashion similar to that of purge, the second process is defined as a function which seeks out specific types of gblock in a sequence and replaces them with their flattened versions - helim for eliminating horvecs, velim for eliminating vervecs:

```
| helim: seq GBLOCK → seq GBLOCK
|------------------------------------------------
| helim ⟨⟩ = ⟨⟩
| ∀ g: dom horvec • helim ⟨g⟩ = bind g
| ∀ g: GBLOCK\dom horvec • helim ⟨g⟩ = ⟨g⟩
| ∀ s, t: seq GBLOCK • helim( s ⁀ t ) = helim( s ) ⁀ helim( t )
```

```
| velim: seq GBLOCK → seq GBLOCK
|------------------------------------------------
| velim ⟨⟩ = ⟨⟩
| ∀ g: dom vervec • velim ⟨g⟩ = bind g
| ∀ g: GBLOCK\dom vervec • velim ⟨g⟩ = ⟨g⟩
| ∀ s, t: seq GBLOCK • velim( s ⁀ t ) = velim( s ) ⁀ velim( t )
```

To specify the third process of merging adjacent line type gblocks we need an auxiliary function which makes one line out of two:

```
==================================================
| (_ join _) : (GBLOCK × GBLOCK) ↠ GBLOCK
|------------------------------------------------
| ∀ a, b, c: dom line |
|    (line( c )).before = (line( c )).after = ⟨⟩ ∧
|    (line( c )).cursor = (bind a) ⁀ (bind b) •
|    a join b = c
==================================================
```

The required function then follows naturally:

```
│ merge: seq GBLOCK → seq GBLOCK
├──────────────────────────────────────────────
│ merge ⟨⟩ = ⟨⟩
│ ∀ g: GBLOCK • merge ⟨g⟩ = ⟨g⟩
│ ∀ s, t: seq GBLOCK; p, q: dom line •
│    merge( s ⁀ ⟨p⟩ ⁀ ⟨q⟩ ⁀ t ) = merge( s ⁀ ⟨p join q⟩ ⁀ t )
│ ∀ s, t: seq₁ GBLOCK | (last s ∉ dom line) ∨ (hd t ∉ dom line) •
│    merge( s ⁀ t ) = merge( s ) ⁀ merge( t )
```

The final stage of normalisation, splitting the processed sequence, is performed by determining the first element of the cursor sequence and then specifying the length of this sequence. To help to do this, we define a function which returns a sequence containing just those elements of another (larger) sequence whose indices in this latter are members of a given set.

```
═[X]══════════════════════════════════
│ (_ ↾ _): (ℙ ℕ × seq X) → seq X
├─────────────────────────────────────
│ ∀ v: ℙ ℕ • v ↾ ⟨⟩ = ⟨⟩
│ ∀ s: seq X • {} ↾ s = ⟨⟩
│ ∀ v: ℙ₁ ℕ; s: seq₁ X; n: ℕ •
│    n ∈ dom s ⇒ {n} ↾ s = ⟨s(n)⟩
│    n ∉ dom s ⇒ {n} ↾ s = ⟨⟩
│    v ↾ s = ({min v} ↾ s) ⁀ ((v\{min v}) ↾ s)
└─────────────────────────────────────
```

Putting the three stages together we obtain the desired specifications:

```
┌─VNorm──────────────────────────────
│ IO
│ ΞWINDOWS
│ VOps
│ XY
│ n: ℕ
├────────────────────────────────────
│    β s'.g = front p
│    γ s'.g = (1..n) ↾ q
│    α s'.g = (n+1 .. #q) ↾ q
│    s'.stack = s.stack
│ where
│    v ≙ (bind⨾ velim⨾ purge) (s.g)
│    p ≙ v LOSPLIT (y, sumhei)
│    q ≙ ⟨ last p ⟩ ⁀ (v HISPLIT (y, sumhei))
└────────────────────────────────────
```

```
┌─HNorm──────────────────────────────
│ IO
│ ΞWINDOWS
│ HOps
│ XY
│ n: ℕ
├────────────────────────────────────
│    β s'.g = front p
│    γ s'.g = (1 .. n) ↾ q
│    α s'.g = (n+1 .. #q) ↾ q
│    s'.stack = s.stack
│ where
│    h ≙ (bind⨾ helim⨾ purge⨾ merge) (s.g)
│    p ≙ h LOSPLIT (x, sumwid)
│    q ≙ ⟨ last p ⟩ ⁀ (h HISPLIT (x, sumwid))
└────────────────────────────────────
```

Section 7: Recursion in the MaCHO Interface

The aim of this document is to formally specify the behaviour of the MaCHO Interface or, more precisely, the effect of certain operations on its underlying data structure. Given the recursive nature of this last item, it is natural that most if not all of the functions to be specified are themselves recursive in one way or another. In this section we describe the general form this recursion takes.

If we were to classify horvecs or vervecs in MaCHO-I according to the number of elements in their cursor sequences, we could choose to distinguish two classes - those with cursors containing two or more gblocks and those with cursors of size exactly one. It is this latter class of gblocks which give rise to the recursion in the interface: in these cases the result of applying a function to the horvec or vervec concerned is determined by the type of the single gblock in its cursor and on whether the function can be recursively applied to this same gblock. In (very) pseudo Pascal terms we have:

```
FUNCTION f (g: GBLOCK): GBLOCK;
BEGIN
   IF                                                   *
      g is a horvec or a vervec AND                     *
      size of cursor = 1 AND                            *
      cursor gblock of g is a valid argument for f      *
   THEN                                                 *
      BEGIN                                             *
         cursor gblock of g := f( cursor gblock of g ); *
         optionally alter g with its new cursor;
         result := g                                    *
      END                                               *
   ELSE
      do something else to g and return result;
END;
```

From the above we see that the recursive call can be followed by further alterations of the input gblock to yield the result; we will come across examples of this behaviour in the document subsequently. In this section, we content ourselves, to begin with, by formally specifying the recursive call part (marked *) for a general function from STATE to STATE:

```
┌─ Recur ──────────────────────────
│ ΞWINDOWS
│ IO
│ FUN
├──────────────────────────────────
│   #(γ s.g) = 1
│   st ∈ dom f
│   β s'.g = β s.g
│   γ s'.g = ⟨result.g⟩
│   α s'.g = α s.g
│   s'.stack = result.stack
│ where
│   │ st, result: STATE
│   ├──────────────────────────────
│   │ st.g = hd (γ[GBLOCK] s.g)
│   │ st.stack = s.stack
│   │ result = f( st )
└──────────────────────────────────
```

```
VRecur ≙ Recur ∧ VOps
HRecur ≙ Recur ∧ HOps
```

(Note that in the informal code we talked about GBLOCKs whereas the formal specification makes reference to STATEs as is, of course, actually required.)

With this general description of recursion, we can easily describe recursive calls of specific functions from the previous section by 'and'ing in the matching schema. For instance, the recursive part of the function (cursor) left applied to a vervec is modelled by:

```
VLeft ≙ VRecur ∧ Left
```

To formally specify a function in full we use the schema override notation ⊕ where for schemas S and T:

```
S ⊕ T ≙ (¬pre T ∧ S) ∨ T
```

This notation models precisely and succinctly the 'if..then..else..' construction of the pseudo code:

```
VFunction ≙ VDoSomethingElse ⊕ (VRecur ∧ Function)
```

and it will therefore be often used in the ensuing text.

For the function ins_char we require a similar recursion scheme to that just presented but one which takes into account the extra character parameter of the ins_char operation:

```
┌─ InsCharRecur ──────────────────────────
│ IO
│ ΞWINDOWS
│ c: Char
├─────────────────────────────────────────
│    #(γ s.g) = 1
│    (st, c) ∈ dom ins_char
│    β s'.g = β s.g
│    γ s'.g = ⟨ result.g ⟩
│    α s'.g = α s.g
│    s'.stack = result.stack
│ where
│   │ st, result: STATE
│   ├─────────────────────────────────
│   │ st.g = hd (γ[GBLOCK] s.g)
│   │ st.stack = s.stack
│   │ result = ins_char( st, c )
└─────────────────────────────────────────

VInsCharRecur ≙ InsCharRecur ∧ VOps
HInsCharRecur ≙ InsCharRecur ∧ HOps
```

Section 8: Going In (to the data structure)

In order to edit text displayed on the screen it must be possible to select the object or objects to be manipulated. In MaCHO-I, an object is selected by positioning the pointer (see section 1) at a certain (x, y) displacement from the top left hand corner of the outermost gblock in a window (see section 4) and "going in". In the interface itself, this corresponds to the action of plummeting down from the outermost gblock through the various levels of structure until a basic gblock, that is, a line or a cartouche, contained therein is found.

When a line type gblock is found at the lowest level, the supplied (x,y) parameters determine a unique element from the sequence of characters within the gblock. The character thus selected becomes the cursor of the line from whence we can define the pre- and post-cursor sequences of the line. However, to go into a line, we need to check the x displacement supplied; if it yields a point beyond the end of the line, we extend the line up to this point with a sequence of space characters. This padding is performed by the function pad:

```
| pad: (seq Char × ℕ) ⇸ seq Char
|---------------------------------------------------------
| ∀ s: seq Char; n: ℕ | n < sumwid s • pad(s, n) = s
| ∀ s: seq Char; n: ℕ | n ≥ sumwid s •
|                         pad(s, n) = pad(s ⁀ ⟨space⟩, n)
```

```
┌─LGoIn──────────────────────────────
│ ΞWINDOWS
│ IO
│ LOps
│ ΞXY
├────────────────────────────────────
│   β s'.g = front p
│   γ s'.g = ⟨ last p ⟩
│   α s'.g = l HISPLIT (x, sumwid)
│   s'.stack = s.stack
│ where
│   l ≙ pad( bind[Char] (s.g), x )
│   p ≙ (l LOSPLIT (x, sumwid[Char]))
└────────────────────────────────────
```

Going into a cartouche is at the same time the most simple and complicated operation in this section. The simple case occurs when the x displacement given lies somewhere along the length of the cartouche for then the result of the operation is merely the same as the input:

```
┌─CGoInSimple──────────
│ ΞWINDOWS
│ IO
│ ΞXY
├──────────────────────
│ s.g ∈ dom cartouche
│ x < wid s.g
│ s' = s
└──────────────────────
```

If on the other hand the x displacement lies somewhere beyond the length of the cartouche, the gblock part of the result is a horvec. This horvec is created from the cartouche and a blank line which extends from the rightmost edge of the cartouche to the selected point:

```
┌─CGoInComplex ──────────────────────────────
│ ΞWINDOWS
│ IO
│ ΞXY
├────────────────────────────────────────────
│   s.g ∈ dom cartouche
│   x ≥ wid s.g
│   s'.g ∈ dom horvec
│   β s'.g = ⟨ s.g ⟩
│   γ s'.g = ⟨ l ⟩
│   α s'.g = ⟨⟩
│   s'.stack = s.stack
│ where
│   l ≙ make[Char]( pad( ⟨⟩, x − wid( s.g ) ) )
└────────────────────────────────────────────
```

CGoIn ≙ CGoInComplex ∨ CGoInSimple

The first part of the definition of goin for a vervec is similar to that for a line, inasmuch as the (x, y) parameters select an object to be the cursor for the structure and from this the rest of the vervec is defined. Obviously though, for a vervec, it is the distance down the structure, the y displacement, which determines the partitioning:

```
┌─VPartition ────────────────────────────
│ ΞWINDOWS
│ IO
│ VOps
│ ΔXY
├────────────────────────────────────────
│   β s'.g = front p
│   γ s'.g = ⟨ last p ⟩
│   α s'.g = v HISPLIT (yy, sumhei)
│   s'.stack = s.stack
│   x' = x
│   y' = yy − sumhei( β s'.g )
│ where
│   v ≙ bind[GBLOCK]( s.g )
│   yy ≙ min {y, sumhei( v ) − 1}
│   p ≙ v LOSPLIT (yy, sumhei[GBLOCK])
└────────────────────────────────────────
```

Note that the displacement used to partition the vervec (yy) is the lower of the maximum possible displacement (the height of the entire vervec minus one) and the original y parameter.

The next and final stage of the definition of goin for a vervec is to recursively "go in" to the gblock in its cursor with the displacement calculated from the partitioning stage:

VGoIn ≙ VPartition ⨟ S

For a horvec, the partitioning part of goin is almost identical to that for a vervec - with the obvious exception that it is the distance along the structure which determines the components of the result and thence it is the widths of the gblocks in the sequence which is important:

```
HPartition
 ΞWINDOWS
 IO
 HOps
 ΔXY
 ─────────────────────────────
    β s'.g = front p
    γ s'.g = ⟨ last p ⟩
    α s'.g = h HISPLIT (xx, sumwid)
    s'.stack = s.stack
    x' = x − sumwid( β s'.g )
    y' = y
 where
    h ≙ bind[GBLOCK] (s.g)
    xx ≙ min {x, sumwid( h ) − 1}
    p ≙ h LOSPLIT (xx, sumwid[GBLOCK])
```

Analogously to vervecs again, the second stage in the defintion of goin for a horvec consists of a recursive "go in" to the cursor gblock: in horvecs, however, this is not the last stage - as it is for vervecs - since we require that the result from the second stage described by schema S to be normalised. The extra processing required is to handle the case when the user selects a point beyond the rightmost end of a horvec whose last element is a cartouche: the resulting "go in" to the cartouche would result in a horvec within a horvec by definition of CGoInComplex. Thus we have:

HGoIn ≙ HPartition ⨾ S ⨾ HNorm

Section 9: Cursor Control Functions

At any stage, we may wish to change the position of the cursor, away from that established by the initial goin. For large movements we can of course execute a further goin at the desired point but if we want only to move to the adjacent character or gblock it is simpler (and more efficient) to use the normal cursor movement commands - cursor left, cursor right, cursor up and cursor down.

To specify such movement, we introduce two schemas - StepForw to move the cursor sequence of a gblock onto the first element of the postcursor sequence and StepBack to move it onto the last element of the precursor sequence:

```
┌─ StepForw ──────────────────
│ IO
│ ΞWINDOWS
├─────────────────────────────
│ β s'.g = (β s.g) ⁀ (γ s.g)
│ γ s'.g = ⟨ hd (α s.g) ⟩
│ α s'.g = tl (α s.g)
│ s'.stack = s.stack
└─────────────────────────────
```

```
┌─ StepBack ──────────────────
│ IO
│ ΞWINDOWS
├─────────────────────────────
│ β s'.g = front (β s.g)
│ γ s'.g = ⟨ last (β s.g) ⟩
│ α s'.g = (γ s.g) ⁀ (α s.g)
│ s'.stack = s.stack
└─────────────────────────────
```

In a vervec, cursor movement up or down takes place in two stages - the first changes the structure according to the relevant "step" schema above and the second recursively "goes in" to the newly defined cursor:

VDown ≙ ((StepForw ∧ VOps)⨾ S) • (VRecur ∧ Down)
VUp ≙ ((StepBack ∧ VOps)⨾ S) • (VRecur ∧ Up)

As a horvec is a horizontal sequence of gblocks, a call to move the cursor up or down it only makes sense if the operations can be recursively applied:

HUp ≙ HRecur ∧ Up
HDown ≙ HRecur ∧ Down

Moving left in a line is easily specified:

LLeft ≙ StepBack ∧ LOps

and moving left in a horvec is defined analogously to moving up in a vervec:

HLeft ≙ ((StepBack ∧ HOps)⨾ S) • (HRecur ∧ Left)

The move left operation in a vervec is defined recursively:

VLeft ≙ VRecur ∧ Left

To shift the cursor of a line to the right is another simple schema:

LRight ≙ StepForw ∧ LOps

but the specifications of rightwards movement for horvecs and vervecs are more complicated as will be seen.

When we apply the right function to a vervec, we transform the cursor of the vervec into a two element horvec. The first element of this horvec is a vervec made from the cursor sequence of the vervec and the second element is a blank line:

```
┌─ VRight1 ──────────────────
│ IO
│ ΞWINDOWS
│ VOps
├────────────────────────────
│    β s'.g = β s.g
│    γ s'.g = ⟨ h ⟩
│    α s'.g = α s.g
│    s'.stack = s.stack
│ where
│    │ h: dom horvec
│    │ v: dom vervec
│    ├──────────────────────
│    │ v = make (γ s.g)
│    │ β h = ⟨ v ⟩
│    │ γ h = ⟨ blank ⟩
│    │ α h = ⟨⟩
└────────────────────────────
```

VRight ≙ VRight1⨟ Set0⨟ S

In a horvec, move right is first of all interpreted as a command to move the cursor onto the first element of its postcursor sequence:

HRight1 ≙ (StepForw ∧ HOps)⨟ S

If, however, the postcursor sequence of the horvec is empty then HRight1 is not defined; in this case a new blank line type gblock is added onto the end of the horvec which is then gone into:

```
┌─ AddLine ──────────────────────────
│ IO
│ ΞWINDOWS
│ HOps
├────────────────────────────────────
│ β s'.g = (β s.g) ⁀ (γ s.g)
│ γ s'.g = ⟨ blank ⟩
│ α s'.g = ⟨⟩
│ s'.stack = s.stack
└────────────────────────────────────
```

HRight2 ≙ AddLine ⨟ Set0 ⨟ S

Although at first glance HRight2 may appear to be an unnatural operation, it is this specification which is the one most used for easily extending the length of horvecs. Altogether then we have:

HRight ≙ (HRight2 ● HRight1) ● ((HRecur ∧ Right) ⨟ HNorm)

Section 10: Inserting Characters

Characters are inserted into lines immediately before the cursor, as may be expected:

```
┌─ LInsChar ──────────────────
│ IO
│ ΞWINDOWS
│ LOps
│ c: Char
├─────────────────────────────
│ β s'.g = (β s.g) ⁀ ⟨ c ⟩
│ ϒ s'.g = ϒ s.g
│ α s'.g = α s.g
│ s'.stack = s.stack
└─────────────────────────────
```

Characters inserted into horvecs are made into line type gblocks and appended to the precursor sequence of the horvec just as a character is added to a line:

```
┌─ InsCharToHor ──────────────────────────────
│ IO
│ ΞWINDOWS
│ HOps
│ c: Char
├─────────────────────────────────────────────
│ β s'.g = (β s.g) ⁀ ⟨ make[Char]( ⟨ c ⟩ ) ⟩
│ ϒ s'.g = ϒ s.g
│ α s'.g = α s.g
│ s'.stack = s.stack
└─────────────────────────────────────────────
```

The result of InsCharToHor must be normalised to merge the newline type gblock with any that are adjacent to it in the horvec:

HInsChar ≙ (InsCharToHor • HInsCharRecur) ⨾ HNorm

In a vervec, the action of ins_char is to transform the cursor of the vervec into a horvec consisting of the line made from the character and a vervec made from the cursor of the vervec:

```
┌─ InsCharToVer ──────────────────────────────
│ IO
│ ΞWINDOWS
│ VOps
│ c: Char
├─────────────────────────────────────────────
│   β s'.g = β s.g
│   γ s'.g = ⟨ h ⟩
│   α s'.g = α s.g
│   s'.stack = s.stack
│ where
│   │ h: dom horvec
│   ├──────────────────────────────────────
│   │ β h = ⟨ make[Char]( ⟨c⟩ ) ⟩
│   │ γ h = ⟨ make[GBLOCK]( γ s.g ) ⟩
│   │ α h = ⟨⟩
└─────────────────────────────────────────────
```

Once the horvec is created in the vervec we go into it:

```
VInsChar ≙ (InsCharToVer ⨾ S) ● VInsCharRecur
```

11.1 Delete Left

For a line, `del_left` has a simple specification:

```
┌─ LDelLeft ──────────────
│ IO
│ ΞWINDOWS
│ LOps
├──────────────────────────
│ β s'.g = front (β s.g)
│ γ s'.g = γ s.g
│ α s'.g = α s.g
│ s'.stack = s.stack
└──────────────────────────
```

For horvecs however, `del_left` does not have such an easily intuitive specification. This complexity arises due to the fact that within a horvec it is possible to delete the character immediately before the cursor using delete left:

```
┌─ BackDelChar ────────────────────
│ IO
│ ΞWINDOWS
│ HOps
├───────────────────────────────────
│    b ∈ dom line
│    # (bind b) > 1
│    β s'.g = front (β s.g) ⁀ ⟨ l ⟩
│    γ s'.g = γ s.g
│    α s'.g = α s.g
│ where
│    b ≙ last (β[GBLOCK] s.g)
│    l ≙ make (front (bind b))
└───────────────────────────────────
```

Obviously there will only be a character to delete if the last element of the precursor sequence of the horvec is a line type gblock. There is yet, though, another complication in that even when there is such a gblock, it must contain at least two characters in order that some characters are left after the deletion. In the cases where there is only one character in the relevant gblock the effect of `del_left` in the horvec is to remove this gblock altogether:

```
┌─BackDelGblock ──────────────
│ IO
│ ΞWINDOWS
│ HOps
├─────────────────────────────
│    b ∈ dom line
│    # (bind b) = 1
│    β s'.g = front (β s.g)
│    γ s'.g = γ s.g
│    α s'.g = α s.g
│    s'.stack = s.stack
│ where
│    b ≙ last (β[GBLOCK] s.g)
└─────────────────────────────
```

Putting these schemas together we have:

HDelLeft ≙ (BackDelChar ∨ BackDelGblock) • (HRecur ∧ DelLeft)

The specification of delete_left for vervecs is somewhat easier:

VDelLeft ≙ (VRecur ∧ DelLeft) ⨾ VNorm

The normalisation stage is required for that case where the cursor of the vervec is a horvec with two elements whose first element is deleted during the action of BackDelGblock: such an action would leave a degenerate horvec in the vervec.

11.2 Delete Right

Like del_left, del_right has a simple effect on lines:

```
┌─LDelRight ──────────────
│ IO
│ ΞWINDOWS
│ LOps
├─────────────────────────
│ β s'.g = β s.g
│ γ s'.g = ⟨ hd (α s.g) ⟩
│ α s'.g = tl (α s.g)
│ s'.stack = s.stack
└─────────────────────────
```

The cursor of the line is deleted and the new cursor is the first character that appeared to the right of the old cursor.

From the specification of LDelRight we notice that the del_right operation is undefined in a line with a null postcursor sequence. When such a situation occurs in a line which is the cursor gblock of a vervec, we replace the cursor sequence of the offending line with a space character and assign this new line to be the cursor of the vervec:

```
┌─VDelPartLine──────────────────────
│ IO
│ ΞWINDOWS
│ VOps
├───────────────────────────────────
│    # (γ s.g) = 1
│    c ∈ dom line
│    β s'.g = β s.g
│    γ s'.g = ⟨ l ⟩
│    α s'.g = α s.g
│    s'.stack = s.stack
│ where
│    c ≙ hd (γ[GBLOCK] s.g)
│    l ≙ make (β c ⁀ ⟨ space ⟩)
└───────────────────────────────────
```

VDelRight ≙ (VDelPartLine⨟ S) ∨ (VRecur ∧ DelRight)

When a line with an empty postcursor sequence forms the cursor of a horvec, and del_right is applied to the horvec, the result is to delete the cursor sequence of the line and move right in the horvec:

```
┌─HDelPartLine──────────────────
│ IO
│ ΞWINDOWS
│ HOps
├───────────────────────────────
│    # (γ s.g) = 1
│    c ∈ dom line
│    bind c ≠ γ c
│    β s'.g = β s.g
│    γ s'.g = ⟨ l ⟩
│    α s'.g = α s.g
│    s'.stack = s.stack
│ where
│    c ≙ hd (γ[GBLOCK] s.g)
│    l ≙ make (β c)
└───────────────────────────────
```

DelRightInHor1 ≙ HDelPartLine⨟ HRight

However, DelRightInHor1 is not defined if the cursor sequence of the line covers the whole line. To handle this case we use DelRightInHor2 which deletes the entire line from the horvec and moves right in the horvec when possible, otherwise the original line is replaced by a blank line:

```
┌─ DelWholeLine ──────────────────
│ IO
│ ΞWINDOWS
│ HOps
├─────────────────────────────────
│   # (γ s.g) = 1
│   c ∈ dom line
│   bind c = γ c
│   β s'.g = β s.g
│   α s.g = ⟨⟩ ⇒
│     (γ s'.g = ⟨ blank ⟩ ∧
│      α s'.g = ⟨⟩)
│   α s.g ≠ ⟨⟩ ⇒
│     (γ s'.g = hd (α s.g) ∧
│     α s'.g = tl (α s.g))
│   s'.stack = s.stack
│ where
│   c ≙ hd (γ[GBLOCK] s.g)
└─────────────────────────────────
```

DelRightInHor2 ≙ DelWholeLine ⨾ Set0 ⨾ S

Altogether we have

DelRightInHor ≙ DelRightInHor1 ∨ DelRightInHor2
HDelRight ≙ DelRightInHor ● (HRecur ∧ DelRight)

Section 12: Grouping Elements

Pervading all of the previous sections is the notion of cursor size and how it affects what is actually performed in the interface. This section and the following one define those operations that allow the displayed cursor to be expanded and contracted in order to achieve precisely the function that the user desires. Such alteration of the cursor has no actual influence on the contents of a gblock and thus there is no need for any post-normalisation stage.

What we do use however is the technique of defining generic schemas to handle horevecs, vervecs and lines in a uniform way just as in Section 2. ExpandBack extends the cursor sequence of the given gblock over the last element of the precursor sequence and ExpandForw extends it over the first element of the postcursor sequence:

```
┌─ ExpandBack ──────────────────
│ IO
│ ΞWINDOWS
├───────────────────────────────
│ β s'.g = front (β s.g)
│ γ s'.g = (last (β s.g)) ⁀ (γ s.g)
│ α s'.g = α s.g
│ s'.stack = s.stack
└───────────────────────────────
```

```
┌─ ExpandForw ──────────────────
│ IO
│ ΞWINDOWS
├───────────────────────────────
│ β s'.g = β s.g
│ γ s'.g = (γ s.g) ⁀ (hd (α s.g))
│ α s'.g = tl (α s.g)
│ s'.stack = s.stack
└───────────────────────────────
```

By supplying the relevant object class to the above two schemas, we achieve the desired specifications:

```
LGroupLeft ≙ ExpandBack ∧ LOps
VGroupLeft ≙ VRecur ∧ GroupLeft
HGroupLeft ≙ (ExpandBack ∧ HOps) ● (HRecur ∧ GroupLeft)

LGroupRight ≙ ExpandForw ∧ LOps
VGroupRight ≙ VRecur ∧ GroupRight
HGroupRight ≙ (ExpandForw ∧ HOps) ● (HRecur ∧ GroupRight)

VGroupUp ≙ (ExpandBack ∧ VOps) ● (VRecur ∧ GroupUp)
HGroupUp ≙ HRecur ∧ GroupUp

VGroupDown ≙ (ExpandForw ∧ VOps) ● (VRecur ∧ GroupDown)
HGroupDown ≙ HRecur ∧ GroupDown
```

Section 13: Ungrouping Elements

In a manner akin to that of the previous section, we can specify most ungrouping actions on horvecs, vervecs and lines using only two schemas - ContractForw and ContractBack make the cursor sequence of the given gblock smaller by paring off its first and last elements respectively. Both schemas are defined only if there are at least two elements in the sequence to be trimmed (since the cursor sequence of a line, horvec or vervec must never be empty, see section 2):

```
┌─ContractForw ──────────────────────
│ IO
│ ΞWINDOWS
├────────────────────────────────────
│ # (γ s.g) > 1
│ β s'.g = (β s.g) ⁀ ⟨ hd (γ s.g) ⟩
│ γ s'.g = tl (γ s.g)
│ α s'.g = α s.g
│ s'.stack = s.stack
└────────────────────────────────────

┌─ContractBack ──────────────────────
│ IO
│ ΞWINDOWS
├────────────────────────────────────
│ # (γ s.g) > 1
│ β s'.g = β s.g
│ γ s'.g = front (γ s.g)
│ α s'.g = ⟨ last (γ s.g) ⟩ ⁀ (α s.g)
│ s'.stack = s.stack
└────────────────────────────────────
```

Ungrouping characters in a line is easily specified:

```
LUngroupLeft ≙ ContractForw ∧ LOps
LUngroupRight ≙ ContractBack ∧ LOps
```

Ungrouping gblocks from the left or the right in a horvec is a more complex operation: if the operation makes the size of the cursor sequence exactly one, we recursively "go in" to this gblock otherwise nothing further is done:

```
HTrimLeft ≙ (ContractForw ∧ HOps)⨟ (S ∨ Null)
HUngroupLeft ≙ HTrimLeft ● (HRecur ∧ UngroupLeft)

HTrimRight ≙ (ContractBack ∧ HOps)⨟ (S ∨ Null)
HUngroupRight ≙ HTrimRight ● (HRecur ∧ UngroupRight)
```

Ungrouping from the top or the bottom in a vervec is analogous to ungrouping from the left or the right in a horvec and so we have:

```
VTrimUp ≙ (ContractForw ∧ VOps)⨟ (S ∨ Null)
VUngroupUp ≙ VTrimUp ● (VRecur ∧ UngroupUp)

VTrimDown ≙ (ContractBack ∧ VOps)⨟ (S ∨ Null)
VUngroupDown ≙ VTrimDown ● (VRecur ∧ UngroupDown)
```

Finally we specify those cases where the direction of the ungrouping required is inappropriate to the orientation of the object structure:

```
HUngroupDown ≙ HRecur ∧ UngroupDown
HUngroupUp ≙ HRecur ∧ UngroupUp
VUngroupLeft ≙ VRecur ∧ UngroupLeft
VUngroupRight ≙ VRecur ∧ UngroupRight
```

Section 14: Inserting Gblocks

As mentioned in section 3, the gblock at the top of the stack of remembered elements can be inserted either before or above the currently displayed cursor. In all cases, the insertion is followed at some stage by "going in" to the newly inserted gblock.

Gblocks can be inserted immediately before the cursor of a line to form a horvec:

```
┌─ InsIntoLine ──────────────────────────────
│ ΞWINDOWS
│ IO
├────────────────────────────────────────────
│ s.g ∈ dom line
│ s'.g ∈ dom horvec
│ β s'.g = ⟨ make( β s.g ) ⟩
│ γ s'.g = ⟨ hd s.stack ⟩
│ α s'.g = ⟨ make( (γ s.g) ⁀ (α s.g) ) ⟩
│ s'.stack = tl s.stack
└────────────────────────────────────────────
```

The result of the insertion is normalised and then re-entered:

```
LInsBefore ≙ InsIntoLine⨾ HNorm⨾ S
```

Inserting a gblock into a horvec at its current cursor position is easily specified:

```
┌─ InsIntoHor ─────────────────
│ ΞWINDOWS
│ IO
│ HOps
├──────────────────────────────
│ β s'.g = β s.g
│ γ s'.g = ⟨ hd s.stack ⟩
│ α s'.g = (γ s.g) ⁀ (α s.g)
│ s'.stack = tl s.stack
└──────────────────────────────
```

```
HInsBefore ≙ (InsIntoHor ● (HRecur ∧ InsBefore))⨾ HNorm⨾ S
```

Inserting a gblock before the cursor sequence of a vervec forms a horvec at that position in a manner similar to that of InsCharToVer:

```
┌─ InsBefIntoVer ──────────────────────────
│ IO
│ ΞWINDOWS
│ VOps
├──────────────────────────────────────────
│   β s'.g = β s.g
│   γ s'.g = ⟨ h ⟩
│   α s'.g = α s.g
│   s'.stack = tl s.stack
│ where
│   │ h: dom horvec
│   ├──────────────────────────────────
│   │ β h = ⟨⟩
│   │ γ h = ⟨ hd s.stack ⟩
│   │ α h = ⟨ make( γ s.g ) ⟩
└──────────────────────────────────────────
```

```
VInsBefore ≙ (InsBefIntoVer ⨾ S) ● (VRecur ∧ InsBefore)
```

Gblocks are inserted above the cursor sequence of vervecs only; for horvecs the operation has to be recursively applied:

```
┌─ InsAboveIntoVer ─────────────────────
│ ΞWINDOWS
│ IO
│ VOps
├───────────────────────────────────────
│ β s'.g = β s.g
│ γ s'.g = ⟨ hd s.stack ⟩
│ α s'.g = (γ s.g) ⁀ (α s.g)
│ s'.stack = tl s.stack
└───────────────────────────────────────
```

```
VInsAbove ≙ (InsAboveIntoVer ● (VRecur ∧ InsAbove))⨾ VNorm⨾ S
HInsAbove ≙ HRecur ∧ InsAbove
```

Section 15: Inserting New Lines

New lines can only be inserted into vervecs but in two ways, either above or below the cursor sequence:

```
┌─ BlankA ──────────────────────
│ IO
│ ΞWINDOWS
│ VOps
├───────────────────────────────
│ β s'.g = β s.g
│ γ s'.g = ⟨ blank ⟩
│ α s'.g = (γ s.g) ^ (α s.g)
│ s'.stack = s.stack
└───────────────────────────────
```

```
┌─ BlankB ──────────────────────
│ IO
│ ΞWINDOWS
│ VOps
├───────────────────────────────
│ β s'.g = (β s.g) ^ (γ s.g)
│ γ s'.g = ⟨ blank ⟩
│ α s'.g = α s.g
│ s'.stack = s.stack
└───────────────────────────────
```

Immediately after a new line has been inserted, it is assumed that the user wishes to edit it and so we "go in" to the new object at its top left hand corner (0,0):

```
VBlankAbove ≙ (BlankA⨾ Set0⨾  S) • (VRecur ∧ BlankAbove)
VBlankBelow ≙ (BlankB⨾ Set0⨾ S) • (VRecur ∧ BlankBelow)
```

For horvecs, blank_above and blank_below are defined recursively:

```
HBlankAbove ≙ HRecur ∧ BlankAbove
HBlankBelow ≙ HRecur ∧ BlankBelow
```

Section 16: Deleting Gblocks

In MaCHO-I, the entire cursor sequence of a horvec or a vervec can be deleted onto the current stack of remembered elements in one operation:

```
┌─Delete1 ─────────────────────────────────
│ IO
│ ΞWINDOWS
├──────────────────────────────────────────
│ α s.g ≠ ⟨⟩
│ β s'.g = β s.g
│ γ s'.g = ⟨ hd (α s.g) ⟩
│ α s'.g = tl (α s.g)
│ s'.stack = ⟨ make( γ s.g ) ⟩ ⁀ s.stack
└──────────────────────────────────────────
```

Delete1 however is not defined if the postcursor sequence of the object gblock is empty - in this case the deletion is performed by Delete2:

```
┌─Delete2 ─────────────────────────────────
│ IO
│ ΞWINDOWS
├──────────────────────────────────────────
│ α s.g = ⟨⟩
│ β s'.g = front (β s.g)
│ γ s'.g = ⟨ last (β s.g) ⟩
│ α s'.g = ⟨⟩
│ s'.stack = ⟨ make( γ s.g ) ⟩ ⁀ s.stack
└──────────────────────────────────────────
```

and so:

$\text{Del} \mathrel{\hat=} \text{Delete1} \vee \text{Delete2}$

Notice that Del is not defined if <u>both</u> the pre- and post-cursor sequences of the object gblock are empty - if it were we would be able to delete the entire input gblock leaving nothing to be recorded as the output.

After the deletion has been performed, we "go in" to the top left hand corner of the new cursor:

$\text{VErase} \mathrel{\hat=} (\text{Del} \wedge \text{VOps}) ⨾ \text{TopVervec} ⨾ \text{Set0} ⨾ \text{S}$
$\text{HErase} \mathrel{\hat=} (\text{Del} \wedge \text{HOps}) ⨾ \text{TopHorvec} ⨾ \text{HNorm} ⨾ \text{Set0} ⨾ \text{S}$

(In a horvec the result of the deletion has to be normalised in case the deletion has given rise to newly adjacent line type gblocks.)

The full specifications of deletion in a horvec and a vervec are thus:

$\text{VDelElement} \mathrel{\hat=} \text{VErase} \bullet (\text{VRecur} \wedge \text{DelElement})$
$\text{HDelElement} \mathrel{\hat=} \text{HErase} \bullet (\text{HRecur} \wedge \text{DelElement})$

Section 17: Duplicating Gblocks

The duplicate function is defined only for horvecs and vervecs, in which case its action is to push a gblock of the respective type onto the top of the given stack of remembered elements. The pushed gblock is composed of the gblocks which are contained in the current cursor sequence of the horvec or vervec to which the operation is applied.

```
┌─ Dup ──────────────────────────────────────
│ IO
│ ΞWINDOWS
├────────────────────────────────────────────
│ s'.g = s.g
│ s'.stack = ⟨ make( ϒ s.g ) ⟩ ^ s.stack
└────────────────────────────────────────────
```

```
VDuplicate ≙ ((Dup ∧ VOps)⨟ TopVervec) ● (VRecur ∧ Duplicate)
HDuplicate ≙ ((Dup ∧ HOps)⨟ TopHorvec) ● (HRecur ∧ Duplicate)
```

Section 18: Undo

The undo function returns the gblock component of the cartouche to which it is applied but only if this gblock is not a line:

```
┌─CUndo ──────────────────
│ IO
│ ΞWINDOWS
├─────────────────────────
│ τ s.g ∉ dom line
│ s'.g = τ s.g
│ s'.stack = s.stack
└─────────────────────────
```

After a cartouche is "undone" we "go in" to the result at some chosen point in order perhaps to "undo" another cartouche therein. However, the result of an undo may be a vervec which, if the original cartouche is itself an element of a vervec, will result in a structure which contravenes the "no X in X" rule of section 3. Similarly, the operation may result in a horvec being created within a horvec and so we have:

```
VUndo ≙ (VRecur ∧ Undo)⨟ VNorm⨟ S
HUndo ≙ (HRecur ∧ Undo)⨟ HNorm⨟ S
```

We now present, in a single unifying block, the full formal definitions of the basic actions within the MaCHO Interface:

```
GOIN ≙ LGoIn v CGoIn v HGoIn v VGoIn
∀ GOIN • goin( s, x, y ) = s'

LEFT ≙ LLeft v HLeft v VLeft
∀ LEFT • left( s ) = s'

RIGHT ≙ LRight v HRight v VRight
∀ RIGHT • right( s ) = s'

UP ≙ HUp v VUp
∀ UP • up( s ) = s'

DOWN ≙ HDown v VDown
∀ DOWN • down( s ) = s'

INS_CHAR ≙ LInsChar v HInsChar v VInsChar
∀ INSERT_CHAR • ins_char( s, c ) = s'

DEL_LEFT ≙ LDelLeft v HDelLeft v VDelLeft
∀ DEL_LEFT • del_left( s ) = s'

GROUP_LEFT ≙ LGroupLeft v HGroupLeft v VGroupLeft
∀ GROUP_LEFT • group_left( s ) = s'

GROUP_RIGHT ≙ LGroupRight v HGroupRight v VGroupRight
∀ GROUP_RIGHT • group_right( s ) = s'

GROUP_UP ≙ HGroupUp v VGroupUp
∀ GROUP_UP • group_up( s ) = s'

GROUP_DOWN ≙ HGroupDown v VGroupDown
∀ GROUP_DOWN • group_down( s ) = s'

UNGROUP_LEFT ≙ LUngroupLeft v HUngroupLeft v VUngroupLeft
∀ UNGROUP_LEFT • ungroup_left( s ) = s'

UNGROUP_RIGHT ≙ LUngroupRight v HUngroupRight v VUngroupRight
∀ UNGROUP_RIGHT • ungroup_right( s ) = s'

UNGROUP_DOWN ≙ HUngroupDown v VUngroupDown
∀ UNGROUP_DOWN • ungroup_down( s ) = s'

UNGROUP_UP ≙ HUngroupUp v VUngroupUp
∀ UNGROUP_UP • ungroup_up( s ) = s'

INS_ABOVE ≙ HInsAbove v VInsAbove
∀ INS_ABOVE • ins_above( s ) = s'
```

```
INS_BEFORE ≙ LInsBefore ∨ HInsBefore ∨ VInsBefore
∀ INS_BEFORE • ins_before( s ) = s'

BLANK_ABOVE ≙ HBlankAbove ∨ VBlankAbove
∀ BLANK_ABOVE • blank_above( s ) = s'

BLANK_BELOW ≙ HBlankBelow ∨ VBlankBelow
∀ BLANK_BELOW • blank_below( s ) = s'

DEL_ELEMENT ≙ HDelElement ∨ VDelElement
∀ DEL_ELEMENT • del_element( s ) = s'

DUPLICATE ≙ HDuplicate ∨ VDuplicate
∀ DUPLICATE • duplicate( s ) = s'

UNDO ≙ CUndo ∨ HUndo ∨ VUndo
∀ UNDO • undo( s ) = s'
```

Section 20: Outer Definitions

At the moment, we have specified the operations of the MaCHO Interface (section 19) without having detailed where their inputs, that is s in schema IO, come from: we shall do this now.

Recall from section 4 that the MaCHO Interface is basically just a set of BRECTs each one of which has a STATE associated with it which defines the gblock and stack of remembered elements for that window. Thus, one can select a STATE by choosing a window and taking the state corresponding to this window:

```
┌─ ChooseState ──────────
│ ΞWINDOWS
│ s': STATE
│ b': BRECT
├────────────────────────
│ b' ∈ dom windows
│ s' = windows( b' )
└────────────────────────
```

Given a scheme whereby we can return a state given a window, it is natural to define another which given a window, replaces the state corresponding to it with a new state:

```
┌─ ReplaceState ──────────────
│ ΔWINDOWS
│ s: STATE
│ b: BRECT
├─────────────────────────────
│ b ∈ dom windows
│ windows' = windows ⊕ {b ↦ s}
└─────────────────────────────
```

Now we are just about ready to specify in full how each operation is carried out in the MaCHO Interface. To do this though, we must specify two things - what happens when the chosen operation can be carried out sucessfully and the error messages which appear when it can't.

The success of an operation is determined by the the success or otherwise of three subtasks:

- selection of an input state
- application of the operation to this input state
- recording of the result of the operation into the interface data structure

The first and last of these subtasks are ChooseState and ReplaceState respectively, the middle task is defined in terms of InnerApply:

```
┌─ InnerApply ─
│ ΞWINDOWS
│ IO
│ FUN
├──────────────
│ s ∈ dom f
│ s'= f( s )
└──────────────
```

We will put these three stages together using as an example the "move cursor left" function.

The middle task is specified by a schema which given <u>any</u> input state returns the state which is the result of moving the cursor left in this state:

LeftOK ≙ InnerApply ∧ Left

The successful application of "move cursor left" on a state extracted from a window in the interface is thus described by:

OLeft ≙ ChooseState ⨟ LeftOK ⨟ ReplaceState

Several other operations that can be performed by the MaCHO Interface are specified in a manner similar to that above:

UpOK ≙ InnerApply ∧ Up
OUp ≙ ChooseState ⨟ UpOK ⨟ ReplaceState

DownOK ≙ InnerApply ∧ Down
ODown ≙ ChooseState ⨟ DownOK ⨟ ReplaceState

DelLeftOK ≙ InnerApply ∧ DelLeft
ODelLeft ≙ ChooseState ⨟ DelLeftOK ⨟ ReplaceState

GroupLeftOK ≙ InnerApply ∧ GroupLeft
OGroupLeft ≙ ChooseState ⨟ GroupLeftOK ⨟ ReplaceState

GroupRightOK ≙ InnerApply ∧ GroupRight
OGroupRight ≙ ChooseState ⨟ GroupRightOK ⨟ ReplaceState

GroupUpOK ≙ InnerApply ∧ GroupUp
OGroupUp ≙ ChooseState ⨟ GroupUpOK ⨟ ReplaceState

GroupDownOK ≙ InnerApply ∧ GroupDown
OGroupDown ≙ ChooseState ⨟ GroupDownOK ⨟ ReplaceState

UngroupLeftOK ≙ InnerApply ∧ UngroupLeft
OUngroupLeft ≙ ChooseState ⨟ UngroupLeftOK ⨟ ReplaceState

UngroupRightOK ≙ InnerApply ∧ UngroupRight
OUngroupRight ≙ ChooseState ⨟ UngroupRightOK ⨟ ReplaceState

UngroupUpOK ≙ InnerApply ∧ UngroupUp
OUngroupUp ≙ ChooseState ⨟ UngroupUpOK ⨟ ReplaceState

UngroupDownOK ≙ InnerApply ∧ UngroupDown
OUngroupDown ≙ ChooseState ⨟ UngroupDownOK ⨟ ReplaceState

UndoOK ≙ InnerApply ∧ Undo
OUndo ≙ ChooseState ⨟ UndoOK ⨟ ReplaceState

Having specified the successful cases above, we now turn out attention to the error cases.

When an error occurs, an error report is signalled to the user. The invocation of such a message in no way affects the contents of the windows in the interface:

REPORT ≙ seq Char

```
┌─Error ────────
│ ΞWINDOWS
│ IO
│ r!: REPORT
└───────────────
```

```
FailLeft ≙ [ Error | r! = "Cannot Move Left" ]
FailUp ≙ [ Error | r! = "Cannot Move Up" ]
FailDown ≙ [ Error | r! = "Cannot Move Down" ]
FailDel ≙  [ Error | r! = "Cannot Delete Object" ]
FailGroupLeft ≙ [ Error | r! = "Cannot Group Left" ]
FailGroupRight ≙ [ Error | r! = "Cannot Group Right" ]
FailGroupUp ≙ [ Error | r! = "Cannot Group Up" ]
FailGroupDown ≙ [ Error | r! = "Cannot Group Down" ]
FailUngroupLeft ≙ [ Error | r! = "Cannot Ungroup Left" ]
FailUngroupRight ≙ [ Error | r! = "Cannot Ungroup Right" ]
FailUngroupUp ≙ [ Error | r! = "Cannot Ungroup Up" ]
FailUngroupDown ≙ [ Error | r! = "Cannot Ungroup Down" ]
FailUndo ≙ [ Error | r! = "Cannot Undo - not a cartouche?" ]
```

By combining the error schemas above with their relevant functions we can specify completely most of the operations of the MaCHO Interface - raising an exception only when the attempted operation fails:

```
OuterLeft ≙ FailLeft ● OLeft
OuterUp ≙ FailUp ● OUp
OuterDown ≙ FailDown ● ODown
OuterDelLeft ≙ FailDel ● ODelLeft
OuterGroupLeft ≙ FailGroupLeft ● OGroupLeft
OuterGroupRight ≙ FailGroupRight ● OGroupRight
OuterGroupUp ≙ FailGroupUp ● OGroupUp
OuterGroupDown ≙ FailGroupDown ● OGroupDown
OuterUngroupLeft ≙ FailUngroupLeft ● OUngroupLeft
OuterUngroupRight ≙ FailUngroupRight ● OUngroupRight
OuterUngroupUp ≙ FailUngroupUp ● OUngroupUp
OuterUngroupDown ≙ FailUngroupDown ● OUngroupDown
OuterUndo ≙ FailUndo ● OUndo
```

We now come to define the 'outer' versions of those functions from section 4 whose appropriate action when the inner application is undefined is <u>not</u> to immediately flag an error (as the preceding defintions of this section have done).

For instance, consider the case where the outer gblock is a cartouche and the user wishes to duplicate this object. Then, since duplicate is defined only for horvecs and vervecs, if we followed the pattern given so far in this section we would signal an error "cannot duplicate" when it would appear that the requested operation is entirely reasonable. Thus we define ODup:

```
┌─ODup ─────────────────────
│ IO
│ ΞWINDOWS
├───────────────────────────
│ s'.g = s.g
│ s'.stack = ⟨ s.g ⟩ ⁀ s.stack
└───────────────────────────
```

and then:

```
DuplicateOK ≙ ODup ● (InnerApply ∧ Duplicate)
ODuplicate ≙ ChooseState; DuplicateOK; ReplaceState
OuterDuplicate ≙ ODuplicate
```

A similar line of argument leads us to define ODel which deletes the outer gblock onto the stack of remembered elements and sets the outer gblock to be a blank line:

```
┌─ ODel ──────────────────────────
│ IO
│ ΞWINDOWS
├─────────────────────────────────
│ s'.g = blank
│ s'.stack = ⟨ s.g ⟩ ⁀ s.stack
└─────────────────────────────────
```

```
DelElementOK ≙ ODel ● (InnerApply ∧ DelElement)
ODelElement ≙ ChooseState⨾ DelElementOK⨾ ReplaceState
OuterDelElement ≙ ODelElement
```

For inserting elements above the outer gblock we define OInsA:

```
┌─ OInsA ──────────────────────────
│ IO
│ ΞWINDOWS
├─────────────────────────────────
│ s'.g ∈ dom vervec
│ β s'.g = ⟨⟩
│ γ s'.g = ⟨ hd (s.stack) ⟩
│ α s'.g = ⟨ s.g ⟩
│ s'.stack = tl (s.stack)
└─────────────────────────────────
```

and for inserting elements alongside the outer gblock we define OInsB:

```
┌─ OInsB ──────────────────────────
│ IO
│ ΞWINDOWS
├─────────────────────────────────
│ s'.g ∈ dom horvec
│ β s'.g = ⟨⟩
│ γ s'.g = ⟨ hd (s.stack) ⟩
│ α s'.g = ⟨ s.g ⟩
│ s'.stack = tl (s.stack)
└─────────────────────────────────
```

After inserting a gblock at the outermost level we normalise the results of the insertion then "go in" to their cursor - just as is done when inserting into a horvec or a vervec:

```
InsAboveOK ≙ (OInsA⨾ VNorm⨾ S) ● (InnerApply ∧ InsAbove)
InsBeforeOK ≙ (OInsB⨾ HNorm⨾ S) ● (InnerApply ∧ InsBefore)
```

All inserts however are only possible if (a) there is something to insert and (b) if there is room to insert the object into the outer gblock (recall the maximum dimensions for gblocks specified in section 5). We can define two error schemas to describe what messages appear when these conditions are violated:

```
NoRoom ≙ [ Error | r! = "No Room to Insert Object" ]
EmptyStack ≙ [ Error | s.stack = ⟨⟩ ∧
                       r! = "No Remembered Elements" ]
InsError ≙ NoRoom ● EmptyStack
```

Then we have:

```
OInsAbove ≙ ChooseState⨟ InsAboveOK⨟ ReplaceState
OuterInsAbove ≙ InsError ● OInsAbove

OInsBefore ≙ ChooseState⨟ InsBeforeOK⨟ ReplaceState
OuterInsBefore ≙ InsError ● OInsBefore
```

We can also, of course, insert a blank line type gblock either above or below the outer gblock:

```
┌─ OBlankB ──────────────
│ IO
│ ΞWINDOWS
├────────────────────────
│ s'.g ∈ dom vervec
│ β s'.g = ⟨ s.g ⟩
│ γ s'.g = ⟨ blank ⟩
│ α s'.g = ⟨⟩
│ s'.stack = s.stack
└────────────────────────
```

```
┌─ OBlankA ──────────────
│ IO
│ ΞWINDOWS
├────────────────────────
│ s'.g ∈ dom vervec
│ β s'.g = ⟨⟩
│ γ s'.g = ⟨ blank ⟩
│ α s'.g = ⟨ s.g ⟩
│ s'.stack = s.stack
└────────────────────────
```

```
BlankAboveOK ≙ (OBlankA⨟ SetO⨟ S) ● (InnerApply ∧ BlankAbove)
BlankBelowOK ≙ (OBlankB⨟ SetO⨟ S) ● (InnerApply ∧ BlankBelow)
OBlankAbove ≙ ChooseState⨟ BlankAboveOK⨟ ReplaceState
OBlankBelow ≙ ChooseState⨟ BlankBelowOK⨟ ReplaceState

OuterBlankAbove ≙ NoRoom ● OBlankAbove
OuterBlankBelow ≙ NoRoom ● OBlankBelow
```

A request to move the cursor right at the outer level creates a horvec whose first element is the original outer gblock and whose second element is a blank line:

```
┌─ OToHor ───────────────
│ IO
│ ΞWINDOWS
├────────────────────────
│ s'.g ∈ dom horvec
│ β s'.g = ⟨ s.g ⟩
│ γ s'.g = ⟨ blank ⟩
│ α s'.g = ⟨⟩
│ s'.stack = s.stack
└────────────────────────
```

This extra blank line can only be inserted if there is room for it and so we have:

```
RightOK ≙ (OToHor⨟ HNorm) ● (InnerApply ∧ Right)
ORight ≙ ChooseState⨟ RightOK⨟ ReplaceState
OuterRight ≙ NoRoom ● ORight
```

Typing a character at the outer level also forms a horvec but again only if there is room:

```
┌─ InsCharApply ──────────
│ IO
│ ΞWINDOWS
│ c: Char
├─────────────────────────
│ (s, c) ∈ dom ins_char
│ s' = ins_char( s, c )
└─────────────────────────
```

```
┌─ OInsC ──────────────────────
│ IO
│ ΞWINDOWS
│ c: Char
├──────────────────────────────
│ s'.g ∈ dom horvec
│ β s'.g = ⟨ make( ⟨ c ⟩ ) ⟩
│ γ s'.g = ⟨ s.g ⟩
│ α s'.g = ⟨⟩
│ s'.stack = s.stack
└──────────────────────────────
```

```
InsCharOK ≙ OInsC ● (InnerApply ∧ InsCharApply)
OInsChar ≙ ChooseState⨾ InsCharOK⨾ ReplaceState
OuterInsChar ≙ NoRoom ● OInsChar
```

For OuterDelRight we need to take into account the case when the outer gblock is a line with a null postcursor sequence since del_right is not defined for this line:

```
┌─ ODelR ─────────────────
│ IO
│ ΞWINDOWS
│ LOps
├─────────────────────────
│ β s'.g = β s.g
│ γ s'.g = ⟨ space ⟩
│ α s'.g = ⟨⟩
│ s'.stack = s.stack
└─────────────────────────
```

```
DelRightOK ≙ ODelR ● (InnerApply ∧ DelRight)
ODelRight ≙ ChooseState⨾ DelRightOK⨾ ReplaceState
OuterInsChar ≙ FailDel ● ODelRight
```

References

[Core87] P.W.Core, "User Extensible Graphics Using Abstract Structure", RSRE Report 87011, August 1987.

[Fabry74] R.S.Fabry, "Capability Based Addressing", CACM 17:7, July 1974, pp. 403-412.

[Foster82] J.M.Foster, I.F.Currie and P.W.Edwards, "Flex: A Working Computer Based on Procedure Values", Proceedings of the International Workshop on High Level Language Computer Architectures, Fort Lauderdale, Florida, December 1982.

[Sufrin86] B.Sufrin, "The Z Handbook", Oxford University Programming Research Group, Draft 1.1, March 1986.

[Spivey88] J.M.Spivey, "The Z Notation: A Reference Manual", to be published by Prentice-Hall International, 1988

[Terry88] P.F.Terry and S.R.Wiseman, "On the Design and Implementation of a Secure Computer Sysem", RSRE Memo 4188, June 1988

[Wiseman88a] S.R.Wiseman, "Protection and Security Mechanisms in the SMITE Capability Computer", RSRE Memo 4117, January 1988.

[Wiseman88b] S.R.Wiseman, "The SMITE Object Oriented Backing Store", RSRE Memo 4147, March 1988.

[Wiseman88c] S.R.Wiseman and H.S.Field-Richards, "The SMITE Computer Architecture", RSRE Memo 4126, January 1988.

DOCUMENT CONTROL SHEET

Overall security classification of sheet UNCLASSIFIED

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference Memo 4247	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known) 7784000	6. Originator (Corporate Author) Name and Location ROYAL SIGNAL & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN, WORCESTERSHIRE WR14 3PS		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title

A Z specification of the MaCHO interface editor.

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials Wood A W	9(a) Author 2	9(b) Authors 3,4...	10. Date 11.1988	pp. ref. 53

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This document describes the basic editor part of the user interface for the SMITE secure computer architecture using the mathematical notation known as Z. Operations that are available to the user, and their effects on the screen display, are specified in conjunction with descriptions of the auxiliary functions and data structures necessary to support them. This specification will be used to implement a powerful yet trustworthy interface for the initiation and control of security related transactions.