Unparsing revisited

Having left the program described in CGN9 behind me, and needing to reimplement it, it seems like a good time to work out the details of the FIFO queues alluded to at the bottom of CGN9-10.

Instead of an infinite sequence of input codes, we use a bounded buffer of them:

$$\text{in} : [0, qn) \rightarrow \text{set of codes}$$

The client of the unparser will be a process that repeatedly inserts codes into this buffer, which will be used as a FIFO queue; the elements of the queue that have been entered by the client but not yet read by the unparser are

$$\text{in}[\text{inp}], \text{in}[\text{inp}+1], \ldots \text{ up to but excluding } \text{in}[qr],$$

where the subscripts are taken mod qn. Note that we explicitly do not rename inp "ql"; because of the possibility that the unparser may back up inp, entries in the buffer to the left of inp may be relevant.

In this respect we observe that neither of the procedures P1 nor P2 contain any occurrences of "!", hence neither of them ever back up. The procedures P3 and PP each contain one occurrence of "!", the left arguments of these occurrences are P2 and P1 respectively. It follows that only <u>one</u> "snapshot of the state" (m1, m2, m3) is relevant at any moment; therefore, m1, m2, and m3 can be stored in global variables instead of local variables. Our interest at the moment is m1, which is the value to which inp will be backed up, if it is backed up. We will arrange that if no state snapshot is relevant — that is, if there is no backup from the current state — then m1 contains the value −1, an invalid index into in. We can now code the routine that clients call to insert a formatting code x into the output stream

```
   Out(x)
≡  acq mu
    in do (qr+1) mod qn = inp ∨ (qr+1) mod qn = m1
          → Wait(mu, nonFull)
       od
     ; in[qr] := x
     ; qr := (qr+1) mod qn
     ; V(nonEmpty)
   end
```

We use a mutex mu and condition variables nonFull and nonEmpty in the obvious way.

The procedure P3 changes in a straightforward way: if an attempt to print a segment of an object using 1-line format succeeds, then m1, which stored the point to which the input would have been backed up if it had failed, is set to −1, since its value is no longer relevant, and the condition variable nonFull is signalled, in case the writer has been blocked by m1:

```
    P3
 ≡  let n, c / n = indent
    in c:Read
     ; do Char(c) → Write(c) ; c:Read
        ▯ c = Setb → PP ; c:Read
        ▯ United(c) → Break(n+off(c)) ; c:Read
        ▯ Bp(c) ∧ ¬United(c)
        →   m1, m2, m3  :=  inp, outp, indent
          ; ( P2 ; m1 := -1 ; V(nonFull)
            ! inp, outp, indent := m1, m2, m3
            ; m1 := -1
            ; Break(n+off(c))
            ; c : Read )
       od
    end
```

(I give ; greater binding power than ! .) Note that there is no need to signal nonFull after the second assignment of -1 to m1, since immediately before that assigment, m1 = inp.

P1 and P2 are the same as before, but the Read procedure must signal nonFull, unless m1 ≠ -1, when reading a character cannot unblock a writer because the value of m1 must precede inp in circular order — more precisely, inp lies in the circular interval [m1, qr).

```
    c : Read
≡  do inp = qr → Wait(mu, nonEmpty) od
 ; c := in[inp]
 ; inp := (inp + 1) mod qn
 ; if m1 = -1 → V(nonFull) [] m1 ≠ -1 → Skip fi
```

The buffering of output is simpler: since Break is called only from P3, it is never called when m1 ≠ -1, hence whenever it is called, the output routine can print the line that the call terminates and discard the characters. Hence we use a buffer out whose first index is zero and whose last index is comfortably large, and write:

```
    Write(c)
≡  out[outp] := c
 ; outp := outp + 1
 ; if c = newline
    → let i | i = 0
        in do i < outp → Print(out[i]) ; i := i+1 od
         ; outp , indent := 0,0
        end
   []  c ≠ newline → indent := indent + 1
   fi
 ; if indent ≤ margin → Skip fi
```

Recall that the last line aborts if the margin overflows, thus backtracking from the snapshotted state.

The procedure PP is similar to P3, but simpler:

```
  PP
≡ m1, m2, m3 := inp, outp, indent
 ; (P1 ; m1 := -1 ; V(nonFull)
   ! inp, outp, indent := m1, m2, m3
   ; m1 := -1
   ; P3)
```

The process that does the formatting executes the command

```
acq mu
 in m1 := -1
  ; qr := 0
  ; inp := 0
  ; outp := 0
  ; indent := 0
  ; P3
end
```

If the call to P3 ever returns, it means the client has sent endb codes for which there are no matching setbs. To get a newline at the top level, a united breakpoint should be used; to get a conditional breakpoint at top level, such as would be used in formatting a paragraph, use an ununited breakpoint.

If there is no legal way that the output can be fit within the margin, then this program aborts. It may be more desirable in this case to allow the long lines in the output; this behavior can be achieved by adding a boolean flag as an additional argument to Write; if the flag is set, then the last command in Write, which aborts on overflowed lines, is skipped. The calls to Write from P1 and P2 do not set this flag; hence they will continue to abort and cause backtracking on margin overflow; but the call to Write in P3 does set the flag, since at that point there is no state snapshot to back up to. Alternately, and preferably, Write can check m1: its last line becomes:

$$\underline{\text{if}}\ \text{indent} \le \text{margin} \lor m1 = -1 \rightarrow \text{Skip}\ \underline{\text{fi}}\,.$$

— Greg Nelson, 20 Jan 84