Infinite Pretty-printing in eXene [*]

Allen Stoughton
Kansas State University
Manhattan, KS 66506, USA
`allen@cis.ksu.edu`
`http://www.cis.ksu.edu/~allen/home.html`

Abstract. We describe the design and implementation of a Standard ML of New Jersey library for the interactive pretty-printing of possibly infinite syntax trees. The library handles elision in a novel way, and is implemented using Concurrent ML and the eXene X Window System toolkit.

1 Pretty-printing with Elision

In the modern approach to pretty-printing, as developed by Oppen [Opp80] and others [Mik81, MCC86, Hug95, Wad98], and featured in Paulson's textbook [Pau96], pretty-printing of abstract syntax is a two-stage process:

- First, an abstract syntax tree is converted into a *pretty-printing tree* that abstractly describes how the syntax tree should be formatted. Pretty-printing trees are made up of strings, potential line-break points, and various kinds of *blocks* [Opp80, Pau96] or *boxes* [Mik81, MCC86, Hug95, Wad98].
- Then, the pretty-printing tree is turned into a sequence of lines that will fit in a window (or file) with a specified width. When a block must be printed at a given indentation-level, the entire block is printed on the current line, if it fits. Otherwise, the block is printed on multiple lines, by converting some or all of the block's potential line-breaks into actual line-breaks.

For example, Figure 1 shows three ways in which the block corresponding to a list might be pretty-printed, depending upon the available line-width. Option (b), in which two block entries are combined on a single line, would be forbidden with some block styles.

If a syntax tree can't be pretty-printed in a window with a given width, then some parts of the tree must be elided, i.e., suppressed. One approach to elision is to elide all subtrees of the syntax tree that are deeper than some level, convert the abbreviated tree to a pretty-printing tree, and then print this tree in the normal way [MCC86]. Unfortunately, with this approach, it is difficult to figure out the best depth at which a given tree should be elided; and, uniform elision at a particular depth may make poor use of the window's width.

[*]Appears in K. Hammond and S. Curtis, editors, *Trends in Functional Programming*, vol. 3, pp. 13–24, Intellect, 2002.

```
[x0, x1, x2]
```

(a)

```
[x0, x1,
 x2]
```

(b)

```
[x0,
 x1,
 x2]
```

(c)

Figure 1: Pretty-printing Options

```
[first,
 ?
 third]
```

Figure 2: Confusing Elision

Another approach is to carry out elision as the pretty-printing tree is converted into a sequence of lines. This approach makes good use of the available window-width. Unfortunately, it can result in a program construct being abbreviated in such a way that some of its top-level keywords or symbols have been elided, which will be ugly and confusing. E.g., some of the commas that separate the elements of a list might be elided, along with the elements they follow; this has occurred in Figure 2, where the ellipsis "`?`" marks the point where the text "`second,`" was elided. What is needed is a way of specifying what the top-level syntax of a block is. An entire block should be elided, unless it is possible for all of its top-level syntax to be printed. The brackets and commas comprise the top-level syntax of the list of Figure 2. Thus, it would be better to pretty-print this list as in Figure 3.

In the pretty-printing trees of Mikelsons [Mik81], one can indicate the top-level syntax of a block by marking certain of its entries as "requiring expansion" (pp. 111–112). A block that is required to be expanded can itself contain certain entries that are required to be expanded, and so on. An entire block will be elided, unless all of its parts that are required to be expanded can be displayed.

Mikelsons's pretty-printing algorithm selects a fragment of a pretty-printing tree to be displayed, where a tree y is a *fragment* of a tree x, iff y can be formed by first selecting a subtree z of x, and then removing certain sub-trees (possibly none) of z. Initially, the fragment is a single node of the pretty-printing tree—typically, one selected by the user. It's not necessarily the tree's root. Then, in a series of steps, controlled by a system of priorities, the current fragment is expanded to include more of the original tree. Interestingly, this expansion process works both *down* from the fragment's leaves, and *up* from its root.

In comparison to the pretty-printing algorithms commonly used today, Mikelsons's algorithm is both complex and inefficient. On the other hand, it is the only existing algorithm that I know about that handles elision nicely.

```
[first,
 ?,
 third]
```

Figure 3: Appropriate Elision

```
datatype flex = Flex of int * int -> text
and      text = Text of (char * flex option) vector vector
```

Figure 4: Flexible Text Values

2 Infinite Pretty-printing in eXene

This paper describes the design and implementation of a Standard ML of New Jersey (SML/NJ) [AM91] library for interactive pretty-printing. The library's method of handling elision is influenced by, but simpler and more efficient than, that of Mikelsons [Mik81]. In fact, the library is capable of pretty-printing infinite syntax trees. The library is implemented using Concurrent ML (CML) [Rep99] and the eXene X Window System toolkit [GR93], both of which are SML/NJ libraries.

In the following sections, we:

- Describe the *flexible text values* into which pretty-printing trees are translated. The associated ML code makes no use of CML/eXene.
- Describe the *pretty-printing trees* themselves. The associated ML code uses CML at one point; it makes no use of eXene.
- Describe the eXene *flexible text widget*, which is used to display flexible text values.
- Show how some example syntax trees of a simple functional language are pretty-printed using the library.

3 Flexible Text Values

In our pretty-printing scheme, pretty-printing trees are translated into flexible text values before being displayed. Figure 4 shows the mutually-recursive datatypes `flex` of *flexible text values* and `text` of *text values*.

A value of type `text` consists of a two-dimensional vector (immutable array) of characters. When a character is an ellipsis, it is labeled by the flexible text value corresponding to an elided tree.

A value of type `flex` consists of a function that formats a syntax tree so that it will fit into a window of a specified size. The function takes in a pair (wid, $maxHt$) and begins formatting the tree so that each line has no more than *wid* characters. If no more than *maxHt* lines are produced, then it returns the lines produced. Otherwise, it returns the first $maxHt - 1$ lines, followed by a line explaining that truncation occurred. In normal use, *maxHt* will be large enough so that no truncation occurs; the user will be able to vertically scroll through the lines produced.

The ML structure (module) for manipulating (flexible) text values also provides:

- Functions for incrementally producing text values efficiently.
- A datatype of *selections*, where a selection describes part of a text value—typically, one that a user has selected. When the text value is displayed, the selected text will be highlighted in some way. There is a function for extracting the selected part of a text value. There is also a function for describing the drawing that must be done when the selection is adjusted.
- A function that describes the drawing that must be done to display a rectangular part of a text value, typically one that the window system wants to redraw, taking account of the current selection.

Although our only source of flexible text values is pretty-printing trees, it would be possible to generate these values in other ways.

4 Pretty-printing Trees

Figure 5 contains the body of the signature (interface) of the pretty-printing structure. The types `block` and `entry`, of blocks and block entries, are abstract types; values of these types can be constructed using the identically-named functions.

A block entry has five fields:

- An integer `space`, which is the number of spaces that should be printed at the beginning of the entry, if the entry is printed on the same line as a previous entry of its block.
- An integer `indent`, which is the number of spaces that should be printed at the beginning of the entry, if the entry is not printed on the same line as a previous entry of its block, i.e., if it is the first entry of its block, or it comes after a break.
- A string `befor`, which is printed after the initial space or indentation.
- A block `block`, which is printed next. If the entry is being printed *without breaks*, then the block is printed without breaks. Otherwise, the block is printed

```
type block and entry

val block : (unit -> bool * entry list) -> block

val entry : {space  : int,
             indent : int,
             befor  : string,
             block  : block,
             after  : string} ->
            entry

val blockToFlex : char * string * block -> FlexText.flex

val empty      : block
val strEntry   : string -> entry
val entryBlock : entry -> block
val strBlock   : string -> block
```

Figure 5: Specification of Pretty-printing Structure

  - at an indentation level of $ind + n +$ `size befor`, where *ind* is the indentation level at which the entry is being printed and n is `space` or `indent`, whichever is being used (as the overall algorithm now stands, n will always be `indent`),
  - and with the knowledge that `size after`$+$ *aft* characters will follow the (final line of the) block without an intervening break, where `after` is the string described below, and *aft* is the number of characters that will come after the entry.

- a string `after`, which is printed last.

When an entry is printed in *abbreviated* form, its block is replaced by an ellipsis (or, when the block would take up no space when printed without breaks, by the null string), but the rest of the entry is printed in the usual way.

A block consists of a list of entries, along with a boolean, *com*, standing for "combine", which specifies whether some of the entries of a block may be printed on the same line, even though the whole block won't fit on a single line. The `block` function takes in a thunk (function with domain `unit`) returning this data, so as to allow for infinite pretty-printing trees. The top-level syntax of a block consists of the `befor` and `after` fields of its entries. Such a block is printed *without breaks* by simply printing each of its entries without breaks. The block is printed normally, at an indentation level of *ind*, and with the knowledge that *aft* characters will follow the block without an intervening break, according to the following rules:

- The block is printed without breaks, if it will fit in the available space.

```
val empty = block(fn () => (false, nil))

fun strEntry s =
      entry{space = 0, indent = 0, befor = s, block = empty,
            after = ""}

fun entryBlock x = block(fn () => (false, [x]))

val strBlock = entryBlock o strEntry
```

Figure 6: Pretty-printing Convenience Functions

- Otherwise, if the top-level syntax of the block won't fit in the available space, i.e., if the block's entries won't fit even in an abbreviated form in which each entry is abbreviated and each entry but the first is preceded by a break (and indented to level *ind*), then the entire block is elided.
- Otherwise, if *com* is false, then each of the block's entries except the first one is preceded by a break (and indented to level *ind*). All entries except the last one will be pretty-printed with the knowledge that no characters will follow them without an intervening break; but the last entry will be pretty-printed with the knowledge that *aft* characters will follow it.
- Otherwise, printing proceeds as in the preceding case, except that as many entries as possible are combined on a given line, as long as each such entry can be printed without breaks. (The printing of the block will take at least one break, though, since otherwise we would be in the first case.)

The function `blockToFlex` takes in an ellipsis character, a truncation message and a block, and produces a flexible text value that will pretty-print the block, marking elisions with the ellipsis character and truncations with the truncation message.

The `empty` block and the remaining functions are provided for convenience, and are defined as in Figure 6 (changing `false` to `true` in these definitions would make no difference).

The distribution of the pretty-printing library includes, as an example, a function that translates the syntax trees of a simple functional language into blocks. For example, the function turns the list expression

```
[x0, x1, x2]
```

into the block of Figure 7. Of course, the value of the `space` field of the first entry of this block is irrelevant. The combine parameter is set to true, since it is not confusing for some of the elements of a list to be combined on a single line.

As another example, the translation function turns the conditional expression

```
if x0 then x1 else x2
```

```
block(fn () =>
            (true,
             [entry{space = 0, indent = 0, befor = "[",
                    block = strBlock "x0", after = ","},
              entry{space = 1, indent = 1, befor = "",
                    block = strBlock "x1", after = ","},
              entry{space = 1, indent = 1, befor = "",
                    block = strBlock "x2", after = "]"}]))
```

Figure 7: Pretty-printing Block of a List Expression

```
block(fn () =>
            (false,
             [entry{space = 0, indent = 0, befor = "if ",
                    block = strBlock "x0", after = ""},
              entry{space = 1, indent = 0, befor = "then ",
                    block = strBlock "x1", after = ""},
              entry{space = 1, indent = 0, befor = "else ",
                    block = strBlock "x2", after = ""}]))
```

Figure 8: Pretty-printing Block of a Conditional Expression

into the block of Figure 8. In this case the combine parameter is false, making it impossible for two parts of a conditional to appear on one line, with the remaining part appearing on another line.

The abstract types `block` and `entry` are implemented by the datatypes of Figure 9. To avoid the repetitive processing of pretty-printing trees, a block consists of a reference to a storage location of type `data`. Initially, the contents of such a location will be a thunk passed to the `block` function. But, when this part of the tree is explored, the thunk will be replaced by the value it returns, annotated by an approximation to its size. A size of `Exactly` m means that the block can be printed, without breaks, using exactly m characters. A size of `AtLeast` m means that the block can't be printed, without breaks, in $m - 1$ or fewer characters.

When the pretty-printing procedure needs to know if a block can be printed, without breaks, using no more than n characters, it begins by consulting the stored size, if it exists. If this size is exact or is `AtLeast` m, where $m > n$, then the question is answered immediately. Otherwise, a better approximation of the block's size must be found by processing the block; this new size will then be stored in the block's storage location. A block is always measured before it is otherwise processed.

Pretty-printing trees are measured using a depth-first, left-to-right, traversal. Unfortunately, this process will fail to terminate for some pretty-printing trees. One reason for this

```
datatype size = Exactly of int | AtLeast of int

datatype block = Block of data ref

and      data  = LazyData of unit -> bool * entry list
               | MeasData of size * bool * entry list

and      entry =
           Entry of
             {space  : int,
              indent : int,
              befor  : string,
              block  : block,
              after  : string}
```

Figure 9: Implementation of Pretty-printing Types

is that the thunks stored in storage locations of type `data` may fail to return. But, since it is legal for the `space` and `indent` fields of an entry to be zero, and for the `befor` and `after` fields to be empty, the depth-first traversal may go on forever without determining that a tree's size exceeds a bound n. It is the programmer's responsibility to avoid creating pretty-printing trees that can't be successfully measured.

Since parts of the same pretty-printing tree may be shared by different windows, and thus operated on by different CML threads, and since the measuring of pretty-printing trees involves the updating of storage locations, care must be taken to stop the different threads from interfering with one another. The flexible text values produced by `blockToFlex` use a global CML lock to avoid interference, during the brief periods when they are generating text values. (There is no point in using finer-grained locking, since such interference will be very rare.)

5 The eXene Flexible Text Widget

When using the eXene X Window System toolkit [GR93], a graphical user interface is constructed out of a hierarchy of widgets. To enable flexible text values to be displayed as part of a GUI, it seemed natural to write a new kind of widget called a *flexible text widget.*

An instance of a flexible text widget is created by:

- creating a vertical scrollbar widget;
- creating a *command channel*;
- starting a *controller thread* in its *initial state*;
- creating an *internal flexible text widget* (IFTW);

- putting the scrollbar widget to the left of the IFTW using a layout widget;
- returning the layout widget plus the command channel.

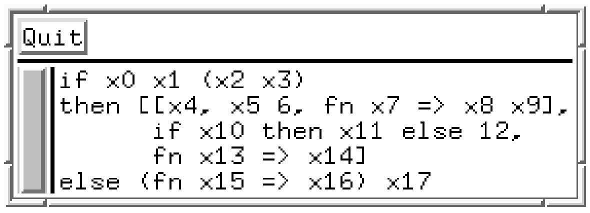
When the controller thread is started, it is given an empty flexible text value to manage, but a function is provided for sending, via the command channel, the controller thread a new value to manage.

When the layout widget is realized, it is given the window that it will manage, along with the window environment via which it will receive mouse and control messages. The layout widget then creates windows for the scrollbar widget and IFTW to manage, positions these windows within its window, creates window environments for these windows, and calls the realize functions of these widgets with the appropriate windows and window environments.

When the realize function of the IFTW is called, it sends the information it is given to the controller thread, via the command channel. We refer to the supplied window as the *parent window*. The controller thread calls its flexible text value with the width of the parent window (in characters, not points) and the maximum usable height of a window. It then creates the *child window* of the parent window in which the resulting text value will be displayed, and enters its *main state*. Typically, the child window will be bigger than the parent window. Only the part of the child window that can be seen through the parent window will be visible to the user. The user will use the scrollbar to scroll the child window within the parent window.

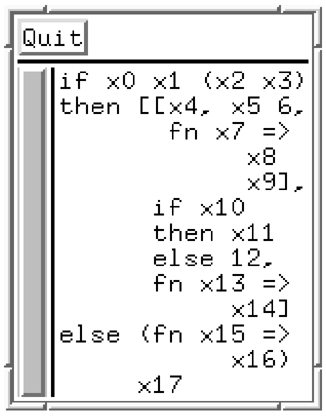
In its main state, the controller responds to:

- **mouse events**, allowing the user to
  - set the X Window System selection, and
  - arrange for elided text to be displayed in new windows, by clicking on ellipses;
- **control events**,
  - redrawing exposed parts of the child window,
  - generating and displaying new text values, when the width of the parent window changes, and
  - entering its final state, when told that one of its windows has been destroyed;
- **scrollbar events**, moving the child window within the parent window, as appropriate;
- **messages from the command channel**, causing new flexible text values to be displayed.

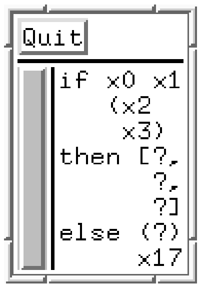


```
if x0 x1 (x2 x3)
then [[x4, x5 6, fn x7 => x8 x9],
      if x10 then x11 else 12,
      fn x13 => x14]
else (fn x15 => x16) x17
```

(a)



```
if x0 x1 (x2 x3)
then [[x4, x5 6,
       fn x7 =>
            x8
            x9],
      if x10
      then x11
      else 12,
      fn x13 =>
           x14]
else (fn x15 =>
           x16)
     x17
```



```
if x0 x1
   (x2
    x3)
then [?,
      ?,
      ?]
else (?)
     x17
```

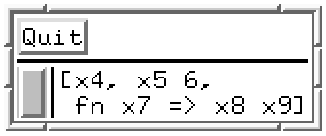
(b) (c)

Figure 10: Views of Finite Expression

6 Pretty-printing Examples

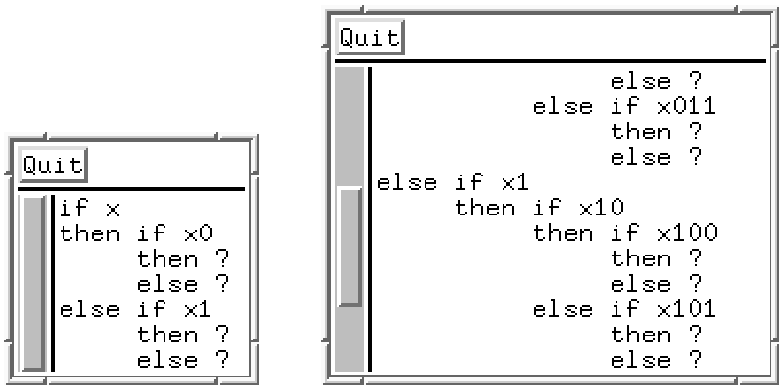
In this section, we show how some example expressions of our simple functional language can be displayed using the pretty-printing library. Figure 10 shows three of the ways that a single conditional expression can be displayed. Each of these examples was produced by bringing up a window containing an initial version of the expression, resizing the window to have the desired width, thus causing the window's contents to be pretty-printed according to that width, and then adjusting the height of the window so that all lines were visible and there were no blank lines (adjusting a window's height won't cause the pretty-printing to be redone). No elision has occurred in the first two cases. In the third case, however, four sub-trees have been elided: the three elements of the list that forms the `then`-part of the conditional, and the anonymous function that occurs in the `else`-part of the conditional. Clicking on the first ellipsis (and then manually resizing the resulting window) produces the contents of Figure 11.

Figure 12 shows two of the ways that an infinite syntax tree with a fairly obvious structure can be pretty-printed. The window of Figure 12(a) displays the first two levels



```
[x4, x5 6,
 fn x7 => x8 x9]
```

Figure 11: Elided Text



```
if x
then if x0
     then ?
     else ?
else if x1
     then ?
     else ?
```

```
                    else ?
               else if x011
                    then ?
                    else ?
else if x1
     then if x10
          then if x100
               then ?
               else ?
          else if x101
               then ?
               else ?
```

(a) (b)

Figure 12: Views of Infinite Expression

of the tree. In contrast, the window of Figure 12(b) displays a vertical section—see the scrollbar—of the first four levels of the tree.

7 Conclusion

The pretty-printing library can be obtained on the WWW at

`www.cis.ksu.edu/~allen/pretty.html`

I am currently using it in the implementation of an operational semantics framework [Sto98], and I hope that many SML programmers will find it useful.

Programming the library using CML and eXene worked out well. CML is well documented and very pleasant to use. Although the eXene toolkit provides a very nice interface to the X Window System, it still has some rough edges and is not completely documented. Several eXene bugs were uncovered during the implementation of my library.

Since the library modules for processing flexible text values and pretty-printing trees don't make use of eXene, it should be possible to interface the library to another windowing system. If the windowing system wasn't concurrent in nature, then the CML lock used by the `blockToFlex` function (see Section 1.4) could be dispensed with. Otherwise, it would have to be replaced by some other kind of lock.

Acknowledgements

It is a pleasure to thank John Reppy for giving me a large amount of very useful advice concerning CML and eXene. I would also like to thank the referees for their comments and suggestions. This work was partially supported by NSF/DARPA grant CCR-9633388.

References

[AM91] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 1991.

[GR93] E. R. Gansner and J. H. Reppy. A multi-threaded higher-order user interface toolkit. In Bass and Dewan, editors, *User Interface Software*, volume 1 of *Software Trends*. Wiley, 1993.

[Hug95] J. Hughes. The design of a pretty printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer-Verlag, 1995.

[MCC86] E. Morcos-Chounet and A. Conchon. PPML: A general formalism to specify prettyprinting. In H.-J. Kugler, editor, *Information Processing 86*, pages 583–590. Elsevier Science Publishers, 1986.

[Mik81] M. Mikelsons. Prettyprinting in an interactive programming environment. In *ACM SIGPLAN SIGOA Symposium on Text Manipulation*, volume 16 of *SIGPLAN Notices*, pages 108–116. ACM Press, Portland, Oregon, June 1981.

[Opp80] D. C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[Rep99] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Sto98] A. Stoughton. An operational semantics framework supporting the incremental construction of derivation trees. In *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[Wad98] P. Wadler. A prettier printer. Bell Labs, Lucent Technologies, 1998.