A Literate Programming Tool for Concurrent Clean

Glenn Strong
Glenn.Strong@cs.tcd.ie

May 15, 2001

Abstract

Literate programming has attracted some interest in the functional programming community. This paper presents a prettyprinting algorithm used in a literate programming tool for the functional language Concurrent Clean, and discusses some of the issues involved in prettyprinting layout based languages.

1 Literate programming

This section introduces the concept of literate programming for those who are unfamiliar with it, and makes some suggestions as to the merits of literate programming. There is some coverage of the topic from the point of view of functional programming, which is the authors primary interest.

1.1 What is literate programming?

Literate programming is a style of programming introduced by Donald Knuth in his book of the same name [2]. The central idea of literate programming is:

> Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.[7]

The exact features of the various literate programming systems vary, but they have a number of common features.

1. They all provide some mechanism for writing both code and documentation in a single file, and extracting the program code in a form suitable for compilation.
2. Most provide some form of automatic indexing of the program code. The tools construct an index of identifiers and code fragments, and generate cross reference information.
3. Most systems provide some mechanism for reordering the program code, so that it can be presented in way the author wishes, rather than in the order which the compiler requires. Tools are used to automatically reorder the code for the compiler.

4. Finally, many systems provide some form of prettyprinting, to improve the visual appearance of the program code.

In practice, literate programming revolves around the idea of writing a program as discrete units of code (called *chunks*, *fragments*, or *scraps*), each of which is accompanied by descriptive text, mathematics and diagrams which describes it's purpose. Tools, generically called *weave* and *tangle* produce respectively a document which describes the program (and includes the source code), and the program source code in a form suitable for compilation. Most literate programming systems use the TEX formatter to produce the documentation.

1.2 The advantages of literate programming

Various advantages are claimed for literate programming. They can be summarised under two main headings.

1.2.1 Improved programs

Most users of literate programming systems claim that they produce better programs when they use literate programming techniques. In general, the reasons for this vary, but include

1. The greater care that is taken when writing each section of the program. Because the programmer is thinking about the *reader* of the program (even if such a reader is completely hypothetical), greater care is taken over each section of code as it is written.
2. The greater freedom allowed by the use of chunks. Sections of code can be isolated without the need to place them within separate functions (procedures, methods, whatever the programming language allows). In general, it does not always make sense to move a small section of code into a separate subroutine, even if the clarity of the program might benefit from having the code isolated. Put simply, it is not always practical to formulate arguments and prepare a subroutine, even if the clarity of the code might benefit from it.
3. The close association between the program code and the specification for what the code should do. In most cases the documentation for each code chunk acts not as a description of what the code *does*, but what it *should do*. When the program does not behave as the documentation says it should, it is the program that is at fault. The division of the program into small chunks (a typical chunk will be about a half-dozen to a dozen lines of code) usually makes it quite simple to identify when a program is not consistent with its specification.

1.2.2 Improved documentation

There is little question that literate programs are better documented than programs produced by more usual means.

1. The documentation exists. This may seem trivial, but many software projects are documented poorly if at all. Once the program has been written, the job is usually seen as done. From the point of view of programming this is correct, of course, but when it becomes necessary to maintain the program the lack of documentation can be a major problem. By forcing the documentation to be developed at the same time as the program itself, literate programming ensures that all code is documented.
2. The program documentation is usually correct. That it, the program documentation usually describes the algorithms and data structures used by the program accurately. The close association between the code and documentation means that the documentation is far more likely to be updated as the code is changed, rather than at some future date (which often never arrives).
3. The program documentation is usually also far more usable than normal documentation. The ability to present documentation side by side with the *actual* code which implements the features being discussed clarifies the situation greatly. Also, the navigation material produced by most literate programming tools (automatic indexing and cross referencing of identifiers, for instance, or hypertext tools) improve comprehension of the program. This is a great boon to program maintenance, as it allows a programmer to trace questionable sections of the program. Returning to a program after an interval of, say, a year makes it clear how useful this is.

1.3 Literate Programming and Functional Programming

Many functional programming systems provide special support for literate programming styles. Most of these systems use some variant of a scheme designed by Richard Bird and Philip Wadler in their language Orwell[6]. This style, variously known as "Bird tracks" or "literate scripts", is supported by various compilers for functional programming languages. The Miranda[1] system supports it, as does the language Haskell (and its derivatives gofer and hugs). This scheme is simpler than the approach described above, in that it does not require separate tools to process the source file. Instead, it relies on an inverted commenting scheme in which only lines which are explicitly marked as containing program code are processed by the compiler, and others are ignored (and therefore may contain any required documentation, including any desired TEX formatting). Although this approach is elegant, in practice it can lack some of the utility of styles based on weave and tangle tools. In particular:

1. There is no explicit naming of code chunks. There is also no facility for code rearrangement. While modern functional languages are mostly free from the need to order code for the convenience

[1] Miranda is a trademark of Research Software

of the compiler, it is not always convenient to separate sections of code for individual discussion. Local definitions (where clauses), for instance, cannot always be conveniently separated from the functions they are local to.

2. The automatic indexing provided by the weave processor is lacking in this style of programming. The lack of an automatic cross reference of functions and types is a noticeable impediment to reading large programs.

3. There are generally no prettyprinting facilities. Although the Miranda system includes a basic code formatting tool which performs keyword underlining, there is no tool for more advanced code printing. A number of literate programming tools are distributed with the Glasgow Haskell compiler suite including a prettyprinter which performs a form of internal alignment based on existing internal alignment in the source code.

Carroll Morgan presents an interesting application of this style of literate programming to produce a system for describing the development of a program by interweaving Orwell code and the refinement calculus [3].

1.4 Prettyprinting

Many literate programming systems provide prettyprinting facilities of varying complexity. The usual arguments advanced in favour of prettyprinting include

1. The program code can be expressed in a more logical form, without the constraints of the standard `ASCII` character set. Examples include the use of standard mathematical symbols for inequalities and logical operations.

2. The improved readability of code which has been properly typeset over code which has been typeset in a limited monospace font.

3. Automatic indentation provides a consistent indentation style, even over programs written by different people, over long periods of time.

4. In particular, note that presenting code which has been properly typeset makes it blend more elegantly into the surrounding documentation. The "special status" of code can be reduced by prettyprinting.

Despite this, prettyprinting remains the most contentious issue within the literate programming community. Some feel that literate programming tools are incomplete without some form of prettyprinting, while others refuse to use tools which force their code to fit a uniform style which they may not agree with. Also, the process of prettyprinting is the most complex part of any set of literate programming tools. The bulk of Knuths `cweave` program, for instance, is devoted to preparing the TEX productions for prettyprinting the source code. Prettyprinters also tend to be highly language dependent,

relying on an intimate knowledge of how the programming language is constructed to do their jobs. All these factors have contributed to producing a series of literate programming tools which provide no prettyprinting at all. Possible the best known of these tools is Normal Ramsey's `noweb` tool.

2 The noweb literate programming tool

The `noweb` tool [4] is one of a number of literate programming tools which can trace their ancestry directly back to Knuth's original `WEB` system. It has a number of advantages, however, over both `WEB` and `CWEB`:

1. `noweb` is language independent. Unlike most literate programming tools, it is possible to use `noweb` with any programming language, and even to mix programming languages within a single file. Sourec code is passed through the standard `noweb` tools essentially unchanged, with only the ordering and chunk-expansion performed. The principle casualty of this approach is the loss of a prettyprinting facility, although the preparation of automatic indices has suffered slightly as well.
2. `noweb` has a much simpler syntax than either of Knuth's tools. Many of the control sequences in both `WEB` and `CWEB` are not required in `noweb`, making `noweb` a much easier tool for beginning and casual literate programmers.
3. The design of `noweb` is highly modular, which allows a great deal of flexibility for the introduction of customised processing of the source.

The `noweb` system is implemented as a series of separate programs, which are linked together using the UNIX pipe operator (ports of `noweb` to non-UNIX platforms provide various means to compose the various programs). This modularity has the immediate advantage of allowing the insertion of extra programs into the processing of the source to perform operations not supplied by the core `noweb` system. A number of additional utilities (called *filters*) are available for `noweb` to perform activities such as automatic indexing of identifiers in several languages. This modularity allows for the implementation of prettyprinting facilities as a separate program which can be used as an optional part of the weaving process. This allows those who are opposed to prettyprinting to avoid having their code processed by the filter.

This modularity is provided by `noweb`'s use of an intermediate representation for a web source file which is more suitable for automatic processing than the original text file. A prettyprinter can be written which simply adds appropriate markup to the code in this intermediate format.

3 A Short Example Web

As an example of the literate programming style used in the `noweb` system, a short program is presented here. In a program of this length, many of the advantages of literate programming are not clear. For those unfamiliar with literate programming, however, this example will help to explain some of the concepts involved. Note that the code in this section has been prettyprinted.

Our selected program uses the sieve of Eratosthenes method to generate prime numbers. The algorithm is well known, and will not be described here in any great detail.

The key step in the sieve of Eratosthenes is to take some prime number, p, and eliminate all of its multiples from consideration.

6a ⟨*eliminate multiples of p* 6a⟩≡ (6e)

```
elim p xs ≡ [x\\x←xs | x mod p≠0]
```

Defines:
`elim`, used in chunk 6b.

If we have a list of numbers, the first of which we know (or assume) to be a prime, then we can generate a list of the remaining primes by removing all multiples of this number from the list, and applying our algorithm to the remaining list (the first element of which will now be a prime). Note the references that have been automatically placed around this chunk of code showing how it relates to the rest of the program.

6b ⟨*generate prime numbers* 6b⟩≡ (6e)

```
sieve [p:rest] ≡ [p:sieve( elim p rest )]
```

Defines:
`sieve`, used in chunk 6c.
Uses `elim` 6a.

The entire list of prime numbers is then generated by

6c ⟨*list of all primes* 6c⟩≡ (6d)

```
( sieve [2...] )
```

Uses `sieve` 6b.

When we run the program, we only want to see a certain number of primes. The computer will generate prime numbers (to the limit of its arithmetic), but we are unlikely to want to see all of them. Here, we shall restrict ourselves to the first hundred primes.

6d ⟨*Top level function* 6d⟩≡ (6e)

```
Start ≡ take 100 ⟨list of all primes 6c⟩
```

Finally, we place the previous code chunks in a module, with appropriate headers as required by the clean compiler.

6e ⟨* 6e⟩≡

```
module Primes
import StdEnv
⟨eliminate multiples of p 6a⟩
⟨generate prime numbers 6b⟩
⟨Top level function 6d⟩
```

4 A prettyprinter

It should be clear by now that a prettyprinting tool to assist with functional programming using the `noweb` system would be useful. In general, prettyprinting can be cleanly separated into two distinct activities.

1. Typesetting special characters in the source code. This includes displaying reserved words in a different font to normal code, replacing some operators and symbols with symbols not normally available in `ASCII` (such as $\leq$ for the less-than-or-equal operator).

2. Making changes to the layout of the code, such as reindenting lines, or breaking long lines of code.

For most cases, changing the appearance of special symbols and reserved words can be done in a language independent way, using a simple table of changes to be made. Changing the layout of code, however, usually requires a great deal of knowledge about the lexical (and sometimes semantic) structure of the programming language in question.

4.1 design goals

The primary design considerations of the prettyprinter described in this document were:

1. It should operate with the `noweb` literate programming tools, preferably as a filter in the weave stage of processing.
2. It should require a bare minimum of changes to the source code of the program being prettyprinted. Many systems, such as `CWEB` require the programmer to place markup in the source code as it is written to indicate preferred forms of prettyprinting. Ideally, however, it will be possible to prettyprint an existing program without making any changes.
3. Additionally, should the user decide to stop using the prettyprinter, it should be easy to do so. No changes to the source code of the program should be required to produce a valid document.
4. Wherever possible the prettyprinter should be language independent.
5. When language dependence is required, the programming language Concurrent Clean 1.1 is the desired language[8].
6. The prettyprinter should be configurable. It should be possible to specify how various parts of the program can be typeset through some simple mechanism.

4.2 Implementation decisions

With the design goals in mind, some more concrete implementation decisions could be made. First, the process of prettyprinting is divided into two parts, as noted above.

1. Typesetting of elements of the source code is performed using a simple lookup table containing two items; a token to be found in the source code, and a string containing TEX formatting commands to replace the token with. Some facility will be provided to load these definitions. This allows different replacement tables to be used for different programming languages.
2. Indentation is performed for the language Clean. This allows analysis of the source code to determine the required indentation to be performed without excessive parsing of the source code.

Mechanisms can also be decided upon at this point to implement the desired features of the prettyprinter.

1. The prettyprinter can be configured while it is processing the document containing the source code. This configuration is performed by "directives" embedded in the document. In order to allow the document to be processed without use of the prettyprinter the directives are hidden within TEX comments.
2. Directives are provided to control all of the major operations of the prettyprinter. In particular, directives are available to define and undefine textual replacements in the source code, and to load large tables of replacements from external files.
3. The default operating mode of the prettyprinter is to assume that code is in the language Clean. This means that the prettyprinter will start by loading definitions for that language, and will attempt to indent code chunks. If another language is used, a directive must be embedded in the source file which will cause the prettyprinter to load a file containing definitions for this language.

Within the constraints of these decisions (some of which were taken to simplify the implementation of the tool), quite a bit can be done to improve the printed appearance of code. Simply performing stage 1 of the prettyprinting process, and selecting some appropriate fonts for the standard appearance of code has a great deal of impact on the appearance of the code chunks. For those cases where indentation is not performed, simply obeying the indentation provided by the user is sufficient.

Unfortunately, for languages like Clean (in general, languages which have some form of layout, or "offside" rule), simply duplicating the whitespace provided by the user is insufficient when proportional fonts are used. This is primarily due to the fact that languages with such a rule impose more complex restrictions on indentation than simply requiring lines to have more or less leading whitespace than preceding lines. Many lines require an amount of indentation which depends on the position of some token in a preceding line. Consider, for example, a function

$$\begin{aligned} f\ \alpha\ \beta \mid \alpha = \beta \quad &\equiv \mathbf{True} \\ \mid \mathbf{otherwise} &\equiv \mathbf{False} \end{aligned}$$

In this function, the indentation of the second function guard is dependent on the position of the guard character in the first line. Similarly, the function right hand sides introduced by the symbol ≡ must be lined up, although their exact position is not known until the width of all the guards has been calculated. This renders some kind of automatic indentation necessary if any prettyprinting is to be done at all.

Note that Clean in fact has two mutually exclusive sets of syntax rules. One uses a layout rule to control scope, the other explicitly delimits scope using brace characters ({}) and semicolons. A simple set of rules can be used to convert between programs written in either style. The language Haskell has a similar set of rules. In practice, most programs are written using the layout rule, and the brace notation is typically reserved for programs which are automatically generated by some form of code generator. This prettyprinter only deals with code written using the layout rule.

4.3 Replacements

A table of replacements is maintained by the prettyprinter which is used to identify 'special case' translations of sections of code. For instance,

`->`	`\rightarrow`	→
`=`	`\equiv`	≡
`>=`	`\geq`	≥

The program code is advanced through by three mechanisms:

1. A built in algorithm which treats some sections of code, such as strings and comments, specially
2. An algorithm which attempts to identify code from the replacement table, and perform the required substitutions
3. A default algorithm which attempts to give basic typographic treatment to all other parts of the program. To avoid errors in the matching of the replacement table, this algorithm uses a heuristic to skip over entire identifiers

Replacements are located in the table using a *longest matching token* policy which attempts to replace as much text as possible. For alphanumeric matches complete identifiers must be matched. For example, if the replacement table contains entries for "alpha" and "alphanumeric", then in the text "alphanumeric a b" the match is "alphanumeric". In the text "alphabetic a b" there is *no* match.

4.4 Indentation

For languages like C no indentation is performed, and whitespace in the source is preserved. If the program has been correctly indented by the programmer this is preserved. This allows the prettyprinter to be used as a simple multi-lingual tool, since indentation can be uniquely determined by the depth of the first token on a line in many imperative languages.

For the language Clean, however, the situation is more complex. Indentation in a layout-sensitive language like Clean relies on more than just the leading whitespace depth on the line. As shown in the previous code fragment, indentation frequently relies on lining up certain tokens which may be embedded in a line.

The indentation scheme used in this prettyprinter is based on the idea of identifying certain significant tokens in the source, and indenting the source code appropriately. A function such as

```
f a b  | a == 0     = b
       | otherwise  = f (a-1) (b*b)
```

can be seen as consisting of three distinct parts from the point of view of indentation.

`f a b`	`| a == 0`	`= b`
	`| otherwise`	`= f (a-1) (b*b)`

Where the material in each of the parts (or "columns") should be left aligned within their columns.

If we ignore the possibility of breaking long lines, then the indentation operation devolves to selecting the correct column for any section of code based on:

1. Special characters which introduce new indentation levels (like the guard character "|").
2. Leading whitespace characters which make a line belong to a certain part of the program (this is the layout rule which gives some semantic meaning to whitespace). For instance, if we rewrite the previous code fragment as

```
f a b  | a == 0     = b
       | otherwise  = f (a-1)
                        (b*b)
```

We have not yet mentioned how *local definitions* within functions are treated. In fact, it should be obvious that they can be handled quite straightforwardly by recursively embedding further columns.

`f a b`	`| a == 0`	`= b`
	`| otherwise`	`= g`
		`where`
		`g` `|True` `=f (a-1) (b*b)`

Note that "where" clauses appear aligned with function right hand sides. This can cause problems with very long local definitions. In many cases, it makes more sense to use the code rearrangement facilities of a system like `noweb` to move such large definitions out of such a code chunk and discuss them separately.

This approach has a superficial resemblance to the well known prettyprinting algorithm [5] which places *indent* and *outdent* operations in the source code. There are some critical differences, however, as this algorithm is implicitly based on "standard" imperative languages.

1. Line breaks are usually treated as being just another whitespace character (i.e. as just a single space character). This is appropriate for free format languages (like modern imperative languages), but is not for layout sensitive languages like Clean where *leading* whitespace (i.e. the first set of whitespace characters on a line) have semantic significance which must be preserved.
2. Most prettyprinting models do not allow indentation to occur *within* a line, which is necessary if we are to line up, for instance, the right hand sides of function equations which have guards.

4.5 Implementation

The most difficult part of the indentation process is identifying which column to place each piece of code in, and what to do with any empty columns which are formed. We shall require only five operations to do this:

movetoguard Which can be used to move from column one to column two, which is used to set guards. If the current column is not column one, then this operation has no effect.

movetorhs Which changes the current column to be the third column, which is used to set function right-hand-sides (and local definitions). This operation has no effect if the current position is column three.

pushtabs Which embeds a new series of columns within column three of the current columns. Until a corresponding poptabs operation is seen, all movement operations will refer to this inner set of columns.

poptabs Which ends an embedded series of columns. The pushtabs and poptabs operations should appear in matching pairs.

endline Which returns the current position to column one, and starts a new row of text. When linebreaks are not being inserted by the prettyprinter, there is no trouble in placing this operation; it appears at each linebreak in the source file, and nowhere else.

Consider now how we can place these operations in a code chunk to produce correct indentation.

Before the first piece of text from the chunk is set, a pushtabs operation should be issued, to create the columnar environment in which the code will be set. The first section of code entered will appear in column one; column one is not left until

1. a guard introduction character (|) is seen. This will cause the current column to be moved to column two. The line position in the source file where the guard character was seen in is also noted. In more recent versions of the Clean language a form of local name re-use has been included, introduced by the `#` character. This should be typeset identically to a guard and is simply treated as an alternative form of guard introduction for the purposes of typesetting.

2. a single equals sign (=) is seen, which introduces a function right-hand-side. This is possible if there is no guard. This causes the column position to be moved to column three, and the line position where the "=" symbol was seen is recorded. Note that it is important that this use of equals is not confused with any of the operators in the Clean language which are partially made from the "=" symbol, such as ":==".
3. The end of line is located.

Following the first line, the amount of leading whitespace on each line is examined, and if it exceeds the position where either the guard character was noted, or the function right-hand-side symbol was noted, then the current position is moved to the appropriate column. To see the need for this, consider the following code fragment:

```
f a b  | a == 0     = b
       | otherwise  =(somefunction a) +
                     (someotherfunction b)
```

The only reliable way to identify this kind of continued expression is to observe the indentation depth of the continued characters, and see which section of the function they belong to, as both left-hand-sides and guards can be continued in this manner.

4.6 Worked example of lineup

Consider an example of how we might embed the operations described above in a sample function. As written by the programmer, the function might appear as

```
func a b | valid b = make_adjustment_function a
                        (\c -> map c [1..b])
         | otherwise = error_message
           where
           error_message = abort "error!\n"
```

Selecting columns for the material on the first line is easy. Note the initial push operation, to create the initial set of columns.

```
<push>func a b <2>| valid b <3>= make_adjustment_function a<nl>
```

The second line can be dealt with by examining the indentation of the first non-whitespace character on the line. As it is as deeply indented as the function right-hand-side on the previous line, it is assumed to be a continuation of this line.

```
                    <3>    (\c -> map c [1..b])<nl>
```

Note that column three is introduced at the appropriate whitespace depth, and not immediately before the first character of the line. This allows a certain amount of further indentation on continued lines, which is desirable. The guard character on the next line is indented much as the first

```
          <2>| otherwise <3>= error_message<nl>
```

The keyword "where" is recognised on the next line, and is placed in column three, even though it is only placed as deeply as the guards.

```
<3>where<push><nl>
```

This keyword also causes a further set of columns to be nested within the right hand side to line up the local definitions. The local definitions are then placed in columns as normal. The columns referred to are now those produced by the pushtabs operation.

```
error_message <3>= abort "error!\n"<nl>
```

If a token is seen which is indented less deeply than the position of the "where" keyword which introduced the nested columns, a pop operation is placed at the end of the previous line.

When producing LaTeX formatting commands to typeset this function, push operations are translated to "tabular" environments, and operations to change columns are translated to the appropriate number of LaTeX column changing commands ("&"). At the end of each code chunk, a sufficient number of pop operations are generated to close any open indentation levels. When the above example is typeset (including appropriate font selection), it appears as

```
func a b|valid b     =make_adjustment_function a
                        (λc → map c [1 ... b])
        |otherwise=error_message
                  where
                  error_message= abort "error!\n"
```

4.7 Not everything is a function

So far, we have seen how we can correctly line up the elements of a function. Although functions play a major role in a language like Clean, they are not the only artifacts to appear in a program. In particular, some kinds of type definition require treatment to be lined up correctly. Algebraic type specifications will require this kind of treatment, as will most record definitions.

Layout of algebraic type specifications can, in fact, almost be controlled using only the rules described above. The alternate cases of an algebraic type, for example,

```
:: Dayoff = Saturday
          | Sunday
          | Holiday
```

can be indented simply be treating the "|" character which introduces the various cases as a guard introduction character, and placing it in column two. To avoid a large blank region on the first line, the "=" character which introduces the first case should also be placed in column two. The need for this can be detected by examining the first token on the line, as "::" can never appear as the first character in a function definition. In fact, lineup is seldom required in these cases, as the programmer has usually lined up cases in an algebraic type correctly anyway.

Record definitions require slightly different treatment. Consider a record such as

```
:: Rec = {
    field :: Int,
    otherfield :: Real,
    lastfield :: String
    }
```

Clearly, it is desirable to line up the various fields. In particular, the *types* of each field should be lined up. This presents a similar problem to lining up various function equations; the typeset width of each field name is not known when the typesetting commands are generated. One solution is to place the introductory material in column one, the field names in column two and the types in column three. This requires treating the "::" character in a field definition similarly to the "=" in a function definition within records, and treating the first non-whitespace character on the line as a guard. When a leading "::" is detected on a line and the first character following the next "=" sign is a left brace, then a special lineup mode is selected in the prettyprinter which does just this. The final result appears as

```
:: Rec = {
   field      :: Int,
   otherfield::Real,
   lastfield  :: String
}
```

Note that the width of column one is not set by the leading material, but by the amount of leading whitespace before the first field.

4.8 Further work

There are a number of shortcomings in the design of the current prettyprinter which should be addressed.

1. Indentation of continued function right-hand-sides and guards sometimes requires some intervention from the user, as it relies on the amount of leading whitespace on the continued line. The amount of whitespace placed in the source code is not always correct to produce attractive layout in the typeset code. Some mechanism to deduce an appropriate amount of extra indentation in continued lines is required.
2. The current scheme for replacements is simple, but makes it irritatingly difficult to typeset expressions like `sqrt (10+x)` as $\sqrt{10+x}$. Some mechanism for specifying more general replacements would be useful. Without writing a full parser for Clean, this cannot be done in general, however, it is possible that using regular expressions to locate replacements would provide enough expressive power for most cases.
3. Automatic line breaking is not currently performed. A scheme for breaking very long lines at appropriate points (again, without building a parser for the language) would be useful.

4. The ability to convert programs written using Cleans "brace-notation" form of programming into correctly indented (but brace free) typeset programs would allow readable documentation of programs produced in this style.
5. The approach taken to lining up records is very simple, and fails when more than one field is specified on each line (as may happen in large records). A more comprehensive approach would scan the record definition to see how many columns are required.

5 Conclusions

Typesetting languages like Clean which use a layout rule is very different to typesetting free-format languages. A new approach is needed in which the concept of "indentation" is replaced with the concept of "lineup" and a tool to prettyprint code using this approach is described[1]. This lineup can be performed using only lexical analysis (the source code does not need to be parsed), and can be completely decoupled from other aspects of prettyprinting (such as font and symbol selection). To perform full prettyprinting, some work is still needed, particularly in the areas of line breaking and expression recognition.

References

[1] Glenn Strong. pp source code and documentation. `http://www.cs.tcd.ie/Glenn.Strong/Software/`.

[2] Donald E. Knuth. *Literate Programming.* CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[3] Carroll Morgan. The refinement calculus, and literate development. In Helmut Partsch Bernard Moller and Steve Schuman, editors, *Lecture notes in computer science, no. 755*, pages 161–182. Springer-Verlang, 1993.

[4] Norman Ramsey. Literate-programming tools need not be complex. Report at `ftp.cs.princeton.edu` in `/reports/1991/351.ps.Z`. Software at `ftp.cs.princeton.edu` in `/pub/noweb.shar.Z` and at `bellcore.com` in `/pub/norman/noweb.shar.Z`. CS-TR-351-91, Department of Computer Science, Princeton University, August 1992. Submitted to *IEEE Software*.

[5] Derek C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and System*, 2(4):465–483, Oct 1980.

[6] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[7] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[8] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean 1.1 Language Report.* University of Nijmegen. Web

page with details at `http://www.cs.kun.nl/~clean`, and PostScript version of the language reference available in the file `ftp://ftp.cs.kun.nl/pub/Clean/Clean11/doc/refman11.ps.gz`. This system is undergoing rapid development, and new versions of the language reference may be available.