The Final Pretty Printer

David Christiansen
Indiana University
davidchr@indiana.edu

David Darais
University of Maryland
darais@cs.umd.edu

Weixi Ma
Indiana University
mvc@iu.edu

Abstract

Widely-used pretty printing libraries are built on assumptions from a previous age of computing that are no longer universally true, such as monospace fonts and batch-mode compilers. Furthermore, they are not extensible, which has led to a plethora of similar libraries. We demonstrate an approach to pretty printing that is independently extensible and supports proportional fonts and interactive interfaces.

1 Introduction

With the wealth of available compile-time information, statically typed functional languages should have the richest interactive programming environments. However, our tools are typically stuck in the world of monospaced text-based terminals and ASCII identifiers. While textual terminals are still an important mode of use that programming tools should support, they should be the floor rather than the ceiling for our ambition. The first step towards good programming tools is good tool-making tools. We present one small part of the solution: an extensible, flexible pretty printing system.

A pretty printer is the inverse of a parser. While a parser converts human-written text into a structured representation, a pretty printer converts a structured representation of data into human-readable text.

Ideally, pretty printers should be specified compositionally, so that pretty printers for different productions in an AST can be written individually and then combined. Oppen [19] described a composable imperative pretty printing algorithm, and classic papers by Hughes [15] and Wadler [25] provide a description of pretty printing in a lazy functional language.

Hughes and Wadler consider the pretty printing problem in the abstract, using these considerations to derive combinator libraries for pretty printing. Since these papers were written, however, it has become clear that some of their simplifying assumptions have become limiting assumptions.

For example, libraries for pretty printing tend to assume that each character occupies a fixed width when rendered. This is not even true for English text when written with a proportionally spaced font, and the vast majority of fonts are proportionally spaced. Features such as kerning, ligatures, and mixed left-to-right and right-to-left scripts, which are now standard in most computing contexts, are likewise ignored.

PL '17, Somewhere
2016. .
DOI:

Another limiting assumption of pretty printing is that text exists only for the understanding of humans, and that once it is displayed, the computer can offer no further assistance. However, programming environments such as those for Lisp and Smalltalk have supported a rich notion of text that is supplemented with information about its meaning, enabling tools and interaction strategies that are not possible for mere text. The interactive environment for Idris, a dependently typed functional language, has been modeled on Lisp and Smalltalk using a pretty printer with a design like ours. Section 2 describes some of the tools that this enables.

At the time of writing, there are 19 pretty printing libraries on Hackage. Seven of these were created by copying and then modifying Daan Leijen's implementation of Wadler's interface (`wl-pprint`), adding extensions such as support for ANSI color codes (`ansi-wl-pprint`), semantic annotations (`annotated-wl-pprint`), or embedded monadic effects (`wl-pprint-extras`). While the ability to improve a program and share these improvements is an advantage of free software, derived libraries do not benefit from miscellaneous improvements made to one another unless active effort is made to port them. Additionally, these extensions cannot be used together. Our pretty printing library is extensible, and each of these libraries could be implemented as an extension on top of it. When possible, it is better to link than to fork.

Because documents are written as programs, rather than as a datatype, we call our library the *Final Pretty Printer*. This name is aspirational as well as descriptive: our intention is that that supporting extensibility, current user interface technology, and a variety of scripts means that it is the last pretty printer that you'll ever need.

Contributions

We make the following contributions to the state of functional pretty printing library design:

- We describe a pretty printing library that supports proportional fonts and non-Roman scripts.
- The correctness of our algorithm is derived solely from the laws governing standard functional programming abstractions, and is stable under extension.
- The library supports semantic annotations, making its output more broadly useful.

2 Challenges and Opportunities

Past pretty printing libraries have been built under a number of assumptions, which are no longer universally true:

- Characters occupy a fixed width on the screen

- The rendering of a particular character, and thus its width, is independent of the surrounding characters
- User interface devices are limited to the display and input of text

Today, graphical displays are ubiquitous and computers are used in many languages. This presents challenges and opportunities.

2.1 Unicode

Software is used all over the world, by speakers of many different languages. The syntax of most programming languages, however, is based primarily on characters from the Latin alphabet. Choices about pretty printing technology that assume that all characters are the same width fail to work in contexts where a variety of scripts are used in the same project, such as a program in which some identifier names are technical terms in a variety of languages.

When represented in a fixed-width font, Latin characters are significantly narrower than they are tall. This choice does not work well for Chinese characters, however, which are wider. Other scripts, like many of those from the Indian subcontinent, realize a sequence of characters as a single glyph, because vowels typically modify the preceding consonant character rather than being drawn separately. In some scripts, such as Sinhala, some combining vowels significantly increase the width of the resulting glyph.

Even if a programming language is only intended for use with English-language identifiers and syntax, it can be useful to support Unicode operators, such as arrows and other mathematical operators, that do not fit well within a fixed-width format. Also, because many beautiful fonts are proportional, a pretty printer that can use them is strictly more useful than one that cannot.

2.2 Interactive environments

The computers on which programmers work today are almost universally equipped with a graphical display and a mouse, touchpad, or touchscreen. While early graphical programming environments such as Smalltalk [12], Lisp machines [18], Self [21, 23], and Nuprl [9] made the most of this, allowing the interactive exploration of a live environment, an image-based model is a challenging basis on which to develop maintainable, reliable, redistributable software. In the retreat from graphical environments to batch processing of plain text, however, something important has been lost.

A *presentation* [7, 18] is a link between an region of program output and the underlying application object that it represents. Presentations are perhaps best known from the user interface toolkit of the thoroughly file-oriented Symbolics Lisp machines, though similar ideas were later implemented by companies such as Lucid [11] as well as in other Lisp environments [17, 20]. In an environment with presentations, a REPL might print a result as usual, but a reference to that result could be obtained for a new evaluation task by clicking on it. Other commands could be performed on the underlying object by interacting with its presentation.

```
data Bool = True
          | False
```
(a) No padding

```
data Bool = True
          | False
```
(b) Right padding

```
data Bool = True
          | False
```
(c) Left padding

```
data Bool = True
          | False
```
(d) Centering

Figure 1. Aligning constructors and padding separators

Presentations are also useful in programming environments that do not separate object identity from object structure. For example, the interactive environment for Idris [3] makes heavy use of presentations. All output from the compiler comes with semantic tags that provide the meaning of each name that occurs in them, and all expressions that are output by the compiler come with a reference to the underlying AST object. This AST object can be used to interact directly with the compiler, for example by normalizing an expression in-place in an error message, by seeing the core language representation of a term output in the REPL, or by showing or hiding implicit arguments.

The Final Pretty Printer supports presentations through a system of semantic annotations. When producing a document from some datatype, it can be annotated with its meaning. Later, meanings can be used both to affect the display of the output (e.g. by using different fonts for top-level definitions and locally-bound variables) and to associate the output with commands relevant to its meaning.

2.3 Rendering to the Web

The Haskell code in this paper is pretty printed with the Final Pretty Printer. Rather than reimplementing the quite complex operations needed to render multilingual text, this pretty printer piggybacks on the massive amount of work put into the text rendering engines of Web browsers. Our Haskell pretty printer, built with the Haskell to Javascript compiler `ghcjs`,[1] runs entirely in a browser. The pretty printer's output was converted to PDF for display in this paper.

As can be seen in Figure 1a, the customary alignment of the = character with the vertical bar separators in datatype definitions becomes more subtle when proportional-width fonts are used. A vertical bar is much narrower than =, which leads to the constructors not starting the same distance from the left margin. To achieve an appealing layout, it is necessary to horizontally pad the documents that represent the vertical bars to the width of the = document, in essence making them monospaced once more. Right and left padding, in Figures 1b and 1c, cause the constructors to be appropriately

[1] Available at the time of writing from https://github.com/ghcjs/ghcjs.

aligned, but we chose to center them with respect to one another (Figure 1d). While text formatting is implemented using CSS rules, spacing and alignment are implemented by inserting HTML elements with explicit widths.

2.4 Interactive documents

A previous pretty printer with semantic annotations was used to write Idris's pretty printer, and work is underway to port Idris to the Final Pretty Printer. While Idris's IDE protocol is editor-independent, the Emacs environment (called simply `idris-mode`) currently makes best use of these features.

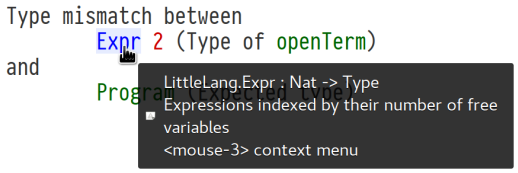
In `idris-mode`, mousing over any identifier provides a tooltip that gives its fully-qualified name, its type, and a summary of its documentation. Right-clicking the name provides a menu that allows queries such as reading the full documentation, getting a list of all definitions that refer to that name or all names that its definition refers to, or browsing other definitions from the namespace in which it is defined.

Additionally, right-clicking any region of compiler output that represents an expression pops up a menu with commands to show or hide implicit arguments, to normalize it, and to show its meaning in Idris's core language. This is especially useful in error messages. Users can interact with the expressions that occur in e.g. unification failures without needing to change a setting and re-provoke the error.

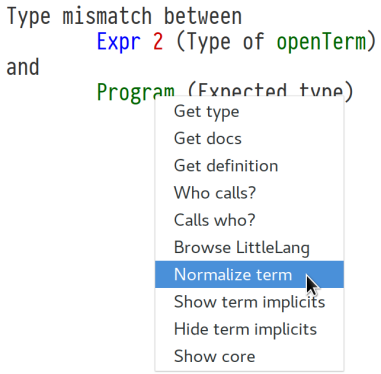
How does all this work? All output from the Idris compiler to an editor has not only a string, but also a list of offset-length-metadata triples that are derived from the annotations. The metadata is a serialization of Idris's internal semantic annotations. While pretty printing, each document representing a name is annotated with its actual name, and each document representing an expression is annotated with that very expression. In other words, these documents *present* their associated objects.

Prior to transmission to the editor, name annotations are enriched with metadata such as documentation and type signatures, and expression annotations are serialized into a form suitable for transmission over a text protocol. Then, editors can use this serialized representation to request other views of the expression, such as the core language or the view in which implicit arguments have been made explicit.

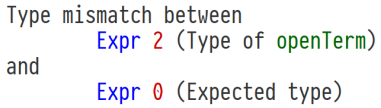
Another advantage of presentations is that they can be used to provide a reference to something for which there is no valid syntax. For example, in systems that have a notion of reference equality and destructive updates, presentations can be used to destructively update a previous REPL value. In systems like Idris, internal names used by the compiler for automatically-generated helper functions do not always have a syntax that the user can type. Inspecting these is much easier when a presentation can be used to indicate which is desired.



(a) Tooltips display metadata for presentations



(b) Contextual menus provide additional commands



(c) Presented terms can be normalized in-place

Figure 2. Interactive error messages in Idris

3 Core Library

A pretty printer constructs a representation of a set of strings, called a *document*, and then arranges for one string from this set to be chosen according to some measure of quality. This set of strings is built from *atomic* strings, which will always occur verbatim in the output, and *conditional* string combinations that can be either be rendered in one line or on multiple lines.

Like many APIs, a pretty printing library can be seen as a domain-specific embedded language [14]. Hughes's and Wadler's pretty printing libraries [15, 25] can be seen as deep embeddings [2] of a pretty printing language in Haskell.

On the other hand, our API can be seen as a shallow embedding of a language similar to that described by Wadler. This approach has the typical advantage of shallow embeddings: the metalanguage can be directly used to extend the embedded language. Traditionally, it is easier to obtain multiple interpretations of deeply embedded languages; we follow the Finally Tagless approach [4] in using type classes to recover multiple interpretations.

3.1 Lines, Widths, and Formatting

In fixed-width contexts, a width is simply a character count. Rendering horizontal space consists of inserting the correct

```
data Chunk w = CText Text
             | CSpace w
  deriving (Eq, Ord)

data Atom w = AChunk (Chunk w)
            | ANewline
  deriving (Eq, Ord)

type Line w fmt = [(Chunk w, fmt)]

data Layout = Flat
            | Break
  deriving (Eq, Ord)

data Failure = CanFail
             | CantFail
  deriving (Eq, Ord)
```

Figure 3. Core datatypes

number of space characters. In order to support proportional-width fonts properly, it is necessary to have a notion of width that is not just a character count. In proportional-width contexts, a width is typically a rational or floating-point number, and rendering horizontal space requires advancing horizontally through some drawing context or constructing an empty box of the appropriate size, because there is no guarantee that the width of space characters evenly divides every horizontal space.

The core datatypes of the library are in Figure 3. Internally, the contents of text lines are represented using the datatype Chunk, which is parameterized over the widths used in the current drawing context.

CText represents a string to be included, and CSpace represents an amount of horizontal space to be skipped. An important invariant is that the text included in CText contains no newlines, and that any space characters included are part of a literal string to be produced that are not opportunities for line breaks.

Different output contexts support different notions of formatted text. A terminal emulator might support some limited font options, such as boldface and colors, while Web browsers and newer TEX implementations such as LuaTEX and XeTEX support the full range of options found in modern OpenType fonts, including customizable ligatures, language-specific glyph variations that share Unicode code points, stylistic sets, and many different weights and sizes.

An individual Line of text is represented as a list of pairs of chunks and formatting options. Lines are not the output of the pretty printer. Rather, they are an intermediate data structure used to track a current line while deciding where to break lines in the output.

The class Measure *w fmt m* in Figure 4 represents a method for determining the horizontal width of a line in some context *m*, where chunks in the line can be formatted using *fmt*. The functional dependencies encode that each rendering context

```
class Measure w fmt m | m → w, m → fmt where
  measure :: Line w fmt → m w
```

Figure 4. The Measure class

has a unique unit of horizontal measurement as well as a unique collection of formatting options.

The widths found in Hughes's and Wadler's pretty printers can be recovered by making formatting trivial, using Int for widths, and measuring by counting characters (*T.length* finds the length of a Text).

```
instance Measure Int () Identity where
  measure = pure ∘ sum ∘ fmap (chunkLength ∘ fst)
    where
      chunkLength (CText t) = T.length t
      chunkLength (CSpace w) = w
```

For the Web backend that was used to render the code in this paper, Measure is more subtle. When pretty printing to a Web page, the precise location of the text in the page's AST can change the text's appearance. Thus, the pretty printer is invoked on a particular target element into which the document is to be displayed. Widths are Doubles that measure the number of horizontal pixels taken up by a document, and formatting consists of a list of strings that represent CSS class names.

Instead of simulating a browser's rendering, measurement in the Web backend is empirical. Measurement consists of adding the formatted text to the output target, and then checking its width once the browser's CSS styles have been applied. A similar technique could be applied to other backends, such as TEX or a drawing canvas.

Rendering that is accurate to fractional pixels motivates a number of design considerations. First, the width of a space is no longer just one unit. The width of a space character can change in different contexts, as determined by the current formatting. Thus, the library provides primitives for measuring the width of a string in the current context.

3.2 Pretty Printing

Rather than use a datatype to represent a document, we instead represent the document directly as a monadic computation that will select a concrete string when run. Not only does this make the pretty printing language extensible, it also enables the re-use of existing Haskell control structures for pretty printing. The type class MonadPretty captures the requirements for a monad to support pretty printing. If there is a MonadPretty *m* instance, then we call *m* a *pretty monad*.

Pretty printing monads are parameterized over types representing widths, semantic annotations, and formatting instructions. These are the parameters *w*, *ann*, and *fmt*, respectively. If the type *w* is to be used for widths, then it must be ordered, numeric, and able to be measured in the pretty monad. Formatting instructions are necessary because

```
class ( Ord w
      , Num w
      , Measure w fmt m
      , Monoid fmt
      , MonadReader (PEnv w ann fmt) m
      , MonadWriter (POut w ann) m
      , MonadState (PState w fmt) m
      , Alternative m
      ) ⇒
      MonadPretty w ann fmt m
      | m → w, m → ann, m → fmt
      where
```

Figure 5. Pretty monads

choices of font might affect the width of a sub-document, and *fmt* must be a monoid so that there is a neutral formatting instruction and so that formatting instructions can be combined.

The actual process of constructing the next line of a concrete string from a document involves backtracking from lines that are too long. Thus, a pretty monad must also implement Alternative. (We discuss backtracking in more detail in Section 3.5.) After each line is completed, it is emitted, so pretty monads must be a MonadWriters of outputs. The line under construction may be read for purposes of measurement, or written to when a new segment fits, so pretty monads are also MonadStates. Finally, information such as the current formatting, the current indentation level, and whether the pretty printer is currently running in flat mode or in line breaking mode has dynamic extent, so pretty monads satisfy MonadReader. The environment also maintains the maximum line width and the ribbon length, which is a maximum width that excludes indentation. The complete definition of MonadPretty is in Figure 5, Figure 6 lists some of the operations that can be derived for any pretty monad, and Figure 7 defines auxiliary datatypes.

3.3 Grouping

Like Oppen's, Hughes's, and Wadler's libraries, the Final Pretty Printer supports *grouping* subdocuments. During rendering, the library should attempt to keep groups on one line, if possible, or place them on multiple lines if they do not fit. When one group contains another, then the inner groups should, if possible, be placed on individual lines even if the entire group does not fit. The Final Pretty Printer uses the Alternative class to implement this backtracking. The entire library maintains the invariant that pretty printing never ultimately fails; all failures are local, and recovered from.

The grouping operator uses the MonadReader operations to communicate with its subdocuments. It first tries to pretty print the subdocument in a context which *disables* line breaks and *allows* failure (*i.e.* calls to `empty`), and if this fails, then tries pretty printing in a context which *enables* line breaks

```
askFormat :: ( MonadReader (PEnv w ann fmt) m
             , Monoid fmt
             ) ⇒
             m fmt

modifyLine :: (MonadState (PState w fmt) m) ⇒
              (Line w fmt → Line w fmt) → m ()

askFailure :: (MonadReader (PEnv w ann fmt) m) ⇒
              m Failure

askMaxWidth :: (MonadReader (PEnv w ann fmt) m) ⇒
               m w

askMaxRibbon :: (MonadReader (PEnv w ann fmt) m) ⇒
                m w

measureCurLine :: ( Measure w fmt m
                  , Monad m
                  , MonadState (PState w fmt) m
                  ) ⇒
                  m w

askNesting :: (MonadReader (PEnv w ann fmt) m) ⇒
              m w
```

Figure 6. MonadPretty operations

```
data PEnv w ann fmt = PEnv { maxWidth :: w
                           , maxRibbon :: w
                           , nesting :: w
                           , layout :: Layout
                           , failure :: Failure
                           , formatting :: fmt
                           , formatAnn :: ann → fmt
                           }

data POut w ann = PNull
                | PAtom (Atom w)
                | PAnn ann (POut w ann)
                | PSeq (POut w ann) (POut w ann)
    deriving (Eq, Ord, Functor)

data PState w fmt = PState { curLine :: Line w fmt }
    deriving (Eq, Ord)
```

Figure 7. Auxiliary datatypes

and *disallows* failure. Because the second attempt disallows failure, it is guaranteed to succeed, maintaining the invariant that pretty printing always produces some answer.

Atomic Chunks are pretty printed using *chunk*, in Figure 8. The first step is to add the chunk to the output, using *tell*, and to append it and its formatting to the current line. Next, *chunk* checks whether there is a failure handler in the dynamic extent of the current document using *askFailure*. If there is, then the current rendering task is speculative, and

```
chunk :: (MonadPretty w ann fmt m) ⇒
         Chunk w → m ()
chunk c =
  do tell $ PAtom $ AChunk c
     format ← askFormat
     modifyLine $ flip mappend [(c, format)]
     f ← askFailure
     when (f == CanFail) $
       do wmax ← askMaxWidth
          rmax ← askMaxRibbon
          w ← measureCurLine
          n ← askNesting
          when (n + w > wmax) empty
          when (w > rmax) empty

grouped :: (MonadPretty w ann fmt m) ⇒
           m a → m a
grouped aM =
  ifFlat aM $ (makeFlat ∘ allowFail) aM <|> aM

ifFlat :: (MonadPretty w ann fmt m) ⇒
          m a → m a → m a
ifFlat flatAction breakAction =
  do l ← askLayout
     case l of
       Flat  → flatAction
       Break → breakAction
```

Figure 8. Determining line breaks

might be undone if it overflows the current line. Thus, the current line is measured, and its width is used together with the indentation level to determine whether adding the chunk was successful.

Previous pretty printing libraries provide a conditional newline operator that may be replaced with a space when inside a group. The Final Pretty Printer decomposes the conditional newline into a more fundamental operation, *ifFlat*, which conditionally selects one or another document depending on whether the current context is flat. Wadler's conditional newline can be recovered as *ifFlat* (*space i*) *newline*, where *i* is the width to use for the space. This decomposition allows newlines to be undone into no space at all. It also allows the library to be used to correctly pretty print languages such as Haskell, where **do**-notation requires semicolons between statements that are on the same line, and Idris, where **case** expressions should use braces and semicolons when on one line, but the layout rule when breaking lines.

Note that a pretty printer computation that always takes the line-breaking branch for each *ifFlat* will never fail. This is because such computations do not contain any grouping, and without grouping, there can be no backtracking. The call to *ifFlat* in *grouped* should be seen as a means of communication with other calls to *grouped* — see Section 7 for a proof that *grouped* is idempotent.

Recovering Hughes-Style Printing Although our pretty printer resembles that of Wadler [25], the original Hughes pretty printing algorithm [15] can also be recovered in our setting by changing the implementation of *grouped*. Hughes-style pretty printing allows inner groups to force line breaks in the context of an outer group. For example, if we redefined *group* to always attempt both sides of the branch, regardless of the inherited context:

```
grouped aM =
  (makeFlat ∘ allowFail) aM <|>
    (makeBroken ∘ disallowFail) aM
```

then we would get the following layout for a nested S-expression:

```
(abd ((a
       b
       c)
      (a
       b
       c)))
```

instead of the current (Wadler-based) algorithm which produces:

```
(abd
 ((a b c)
  (a b c)))
```

For more discussion on the differences between Wadler-style and Hughes-style pretty printing, and for the origin of this example, see Bernardy [1].

3.4 Indentation

Indentation in the Final Pretty Printer is tracked as part of the dynamic environment of pretty monad computations. The current indentation level is used to determine whether a line exceeds the ribbon width, and also to insert space at the beginning of a new line. The *nest* operator increments this level in its dynamic extent, causing indentation levels to follow the lexical structure of the programs being displayed.

Oppen's, Hughes's, and Wadler's pretty printing libraries cause the elements of groups to have the same base indentation. In other words, when the newlines in a group cannot be undone, then the group's members are left-aligned. This is good for expression-oriented programming languages, and also for many block-structured languages. However, popular styles in languages like Javascript and Haskell sometimes call for subgroups to be indented a fixed amount, rather than aligned with their first token. For example, in the following snippet, the contents of the callback are not indented relative to the `function` keyword.

```
window.setTimeout(function () {
  console.log("Message");
}, 5000);
```

Likewise, some Haskell users prefer a "dangling **do**" style, where **do** introduces a block rather than an expression. The do-expression is grouped, but when rendering on more than

one line, those lines should be far to the left of the beginning of the expression.

```
measureText txt = do
  format <- askFormat
  measure [(CText txt, format)]
```

To make both expression-style and block-style printing possible, the Final Pretty Printer provides an *align* operator that increases the nesting to the current column, and an operator *expr* that composes alignment and grouping.

3.5 Instantiating the Interface

So far, we have only discussed the *interface* required to instantiate our pretty printer, embodied in the MonadPretty type class shown in Figure 5. To execute the pretty printer, one must construct a suitable monad which adheres to the MonadPretty interface. We construct this monad using monad transformers [16], for which there are two degrees of freedom in constructing a suitable monad:

1. Choosing a monad to implement Alternative; and
2. Choosing the order of each monad transformer.

For (1) we choose to implement the Alternative interface with Maybe, which has the effect of attempting each branch of pretty printing logic until the first success, after which later branches are not considered.

For (2) we choose the following order of transformers:

```
type DocM w ann fmt a =
  RWST (PEnv w ann fmt)
    (POut w ann)
    (PState w fmt)
    Maybe
    a
```

which induces a datatype equivalent to:

```
type Doc w ann fmt =
  PEnv w ann fmt →
  PState w fmt →
  Maybe (PState w fmt, POut w ann)
```

This ordering ensures that when backtracking occurs, modifications to the state and output are discarded before attempting the next branch of pretty printing. An ordering of RWST and MaybeT in the other direction would have an opposite, undesirable effect.

An Alternative Alternative Another choice for (1) would be to use the list monad ([]) instead of Maybe. This choice of monad stack allows for ranking multiple successful pretty printer renderings in terms of some metric of quality. Our implementation which uses Maybe is ultimately greedy, and always selects the first successful pretty printing branch without attempting any others. This has the defect of not finding "prettier" documents which result from successful layouts which are not lexicographically first in the branching logic of the algorithm. See Bernardy [1] for an effective solution to this problem based on ranking layouts with a quality metric.

4 Semantic Annotations

Semantic annotations cannot be added to a document after it is rendered because, in many cases, the meaning of a document should inform the way that it looks. For example, rendering keywords in bold or variables that refer to types in italic can change the width of that document, so the pretty printer must be able to make use of the relationship between semantics and appearance when rendering. A function from annotations to formatting is provided as part of the pretty printing environment, that is, the Reader portion of the state, and is used to format the output as it is being printed.

Annotations are added using the *annotate* operator. During rendering, *annotate* adds the formatting associated with the desired annotation. The resulting linearized representation of the document contains annotations around some of the substrings, and later output can use the annotations to construct an interface above and beyond the formatting instructions.

There is a caveat: if the formatting applied to annotated documents during rendering does not accurately capture the formatting used when drawing the output, then decisions made on the basis of the widths of subdocuments will not be accurate. We leave the maintenance of this invariant to users.

Running a pretty printing computation results in a POut, which is defined in Figure 7. The PAnn constructor associates meanings with sub-regions of the output, and different interfaces are free to use that information. For instance, it can be used to select ANSI color codes for rendering to a console, or to associate regions in a GUI with their meanings. Because POut *w* is a functor, it is also possible to post-process pretty-printer output to enrich the annotations with information that was not available at pretty-printing time.

5 Extensions

A pretty printing library will not be all things to all people. To support additional features, the Final Pretty Printer can be enriched with additional effects using monad transformers. These extensions can only be used together, however, if they do not change each others' semantics, and the effects are independent. Fortunately, this is the case.

Definition 5.1. A *transformer of pretty monads* is a monad transformer [16] that preserves the specification of the pretty printing operations.

Theorem 5.2. *Every monad transformer is a transformer of pretty monads.*

Proof. If T is a monad transformer, then *lift* commutes with >>=, which implies that the structure of the transformed monads is preserved. The proofs (in Section 7) that our

algorithm satisfies its specification make use only of the abstract laws of their implementation monad, so they also hold for T *m*. □

Two convenient extensions that are widely useful are *variable environments* and *precedence and associativity*.

5.1 Variable Environments

An almost universal feature of programming languages is binding structure, and the scoping rules of a language are not always easy to discover from its surface syntax. Pretty printers with semantic annotations can use the language implementation's facilities for resolving variable scopes to provide an implementation that connects binding sites with use sites. Additionally, bound variables can be annotated with type information, documentation, and other metadata that may not be immediately apparent to readers, and the context in which the document is displayed can then use this information to increase the understandability of the output.

Because the structure of a lexical environment corresponds closely to the structure of an AST, a pretty printer that traverses this structure to produce a document can faithfully represent the lexical environment using an additional MonadReader effect.

To avoid conflicts with the built-in MonadReader constraint on MonadPretty, a wrapper is needed. We begin by defining MonadReaderEnv, which represents readers of some environment *env*.

```
class MonadReaderEnv env m | m → env where
  askEnv :: m env
  localEnv :: (env → env) → m a → m a
```

A monad that is both pretty and a reader of environments is a MonadPrettyEnv.

```
class ( MonadPretty w ann fmt m
      , MonadReaderEnv env m
      ) ⇒
      MonadPrettyEnv env w ann fmt m
      | m → w, m → ann, m → fmt, m → env
      where
```

A **newtype** wrapper around ReaderT is sufficient to define EnvT as a transformer of pretty monads.

5.2 Operator Precedence and Associativity

A reader effect for an environment consisting of operator precedence and associativity information as well as the surrounding context's precedence and associativity can be defined using a construction similar to EnvT. This additional structure can be used to enrich the pretty printing language with operations for inserting parentheses as necessary.

The precedence environment consists of a current level, whether or not it is bumped, and left and right parentheses with optional annotations. "Bumping" a precedence level is used as a tiebreaker to implement associativity for nested applications of operators with the same precedence. The

```
data PrecEnv ann = PrecEnv { level :: Int
                           , bumped :: Bool
                           , lparen :: (Text, Maybe ann)
                           , rparen :: (Text, Maybe ann)
                           }
```

Figure 9. The precedence environment

extension provides an operator *atLevel* that conditionally inserts parentheses depending on the environment's level and a provided level.

6 Performance

The performance of our pretty printing algorithm is comparable to Wadler's. In the best case, the time complexity for pretty printing is $O(n)$ for n atoms in document being printed. A pathological worst case for our algorithm occurs when every atom in the document appears nested inside a group expression. In this worst case, the time complexity for pretty printing is $O(nw)$ for n atoms in the document being printed and w as the maximum layout width. Informally this worst case scenario plays out as follows: at each atom, an attempt is made to format the document on a single line ($O(w)$ work); at this point the current layout branch fails and the algorithm backtracks to introduce a newline after the first atom; and this is repeated for every atom in the document, hence $O(nw)$ work.

Notably, our implementation is *not* sensitive to the strictness of the implementation language, as is the case for both Hughes and Wadler.

7 Correctness

While it is important that software in general be correct, the nature of the correctness argument for the Final Pretty Printer is of special importance. We show that the correctness of the core of the implementation is derived entirely from the laws governing the control structures that are employed, rather than just a specific instantiation of these structures. This allows *arbitrary* monad transformers to be used to extend the library, hopefully putting an end to the forking of pretty printing libraries.

7.1 Prior Work

The original pretty printing paper by Hughes [15] considered the laws that a minimal pretty printing api should satisfy, and derived an algorithm which satisfied those laws. The interface consisted of document construction operators:

(⋄) :: Doc → Doc → Doc	Horizontal concatenation
($$) :: Doc → Doc → Doc	Vertical concatenation
text :: String → Doc	Literal text
nest :: Int → Doc → Doc	Document nesting

and laws which they should satisfy (only some of which we display here):

$$
\begin{aligned}
x \diamond \mathit{text}\ "" &= x \\
(x \diamond y) \diamond z &= x \diamond (y \diamond z) \\
\mathit{text}\ s \diamond \mathit{text}\ s' &= \mathit{text}\ (s +\!\!+ s') \\
(x \mathbin{\$\$} y) \mathbin{\$\$} z &= x \mathbin{\$\$} (y \mathbin{\$\$} z) \\
(x \mathbin{\$\$} y) \diamond z &= x \mathbin{\$\$} (y \diamond z) \\
\mathit{nest}\ 0\ x &= x \\
\mathit{nest}\ k\ (\mathit{nest}\ k'\ x) &= \mathit{nest}\ (k + k')\ x \\
\mathit{nest}\ k\ (x \diamond y) &= \mathit{nest}\ k\ x \diamond \mathit{nest}\ k\ y \\
\mathit{nest}\ k\ (x \mathbin{\$\$} y) &= \mathit{nest}\ k\ x \mathbin{\$\$} \mathit{nest}\ k\ y
\end{aligned}
$$

Later work by Wadler [25] modified the combinator interface:

$\mathit{nil} :: \mathrm{Doc}$	The empty document
$(\diamond) :: \mathrm{Doc} \to \mathrm{Doc} \to \mathrm{Doc}$	Horizontal concatenation
$\mathit{text} :: \mathrm{String} \to \mathrm{Doc}$	Literal text
$\mathit{line} :: \mathrm{Doc}$	Newline
$\mathit{nest} :: \mathrm{Int} \to \mathrm{Doc} \to \mathrm{Doc}$	Document nesting
$\mathit{group} :: \mathrm{Doc} \to \mathrm{Doc}$	Grouping
$(<\!\mid\!>) :: \mathrm{Doc} \to \mathrm{Doc} \to \mathrm{Doc}$	Nondeterminism
$\mathit{flatten} :: \mathrm{Doc} \to \mathrm{Doc}$	Undo line breaks

and presented alternative laws (only some of which we display here):

$$
\begin{aligned}
x \diamond \mathit{nil} &= x \\
\mathit{nil} \diamond x &= x \\
(x \diamond y) \diamond z &= x \diamond (y \diamond z) \\
(x <\!\mid\!> y) <\!\mid\!> z &= x <\!\mid\!> (y <\!\mid\!> z) \\
(x <\!\mid\!> y) \diamond z &= (x \diamond z) <\!\mid\!> (y \diamond z) \\
x \diamond (y <\!\mid\!> z) &= (x \diamond y) <\!\mid\!> (x \diamond z) \\
\mathit{nest}\ 0\ x &= x \\
\mathit{nest}\ i\ (\mathit{nest}\ i'\ x) &= \mathit{nest}\ (i + i')\ x \\
\mathit{nest}\ i\ (x <\!\mid\!> y) &= \mathit{nest}\ i\ x <\!\mid\!> \mathit{nest}\ i\ y
\end{aligned}
$$

To construct a verified pretty printer, Danielsson [10] formalized a variant of Wadler's algorithm in Agda and was successful in proving many of these laws. In the process, it was discovered that the following law from Wadler [25] actually doesn't (and shouldn't) hold:

$$(x <\!\mid\!> y) \diamond z = (x \diamond z) <\!\mid\!> (y \diamond z)$$

7.2 Our Work

The core layout algorithm of our pretty printer is *chunk*, in Figure 8. We use *chunk* to implement higher-level operators for creating and combining documents, a subset of which coincide with Wadler's interface. Our implementation of Wadler's combinators also obeys Wadler's pretty printing laws. The challenge in our setting is how to prove that they hold for *any* pretty monad.

Our pretty printer is parameterized by a monad m which implements various monadic reader, writer, state, and nondeterminism operations. A pretty printer document is then a monadic action $m\ ()$ in this arbitrary monad, and we would like to prove that pretty printing laws hold for *any* m that *chunk* is executed in, up to possibly some restriction on m. The result we describe in this section is that, not only is it *possible* to that our algorithm satisfies the pretty printing laws with respect to an uninstantiated law-abiding monad, but also that the proofs proceed directly from the monad laws.

7.3 Monad Laws

The monad laws which we assume are standard [24]:

$$
\begin{aligned}
\mathit{return}\ x \gg\!= k &= k\ x \\
c \gg\!= \mathit{return} &= c \\
(c \gg\!= f) \gg\!= k &= c \gg\!= \lambda x \to f\ x \gg\!= k
\end{aligned}
$$

We additionally assume laws for nondeterminism:

$$
\begin{aligned}
\mathit{empty} \gg\!= k &= \mathit{empty} \\
c \gg\!= \lambda\_ \to \mathit{empty} &= \mathit{empty} \\
(c1 <\!\mid\!> c2) <\!\mid\!> c3 &= c1 <\!\mid\!> (c2 <\!\mid\!> c3) \\
(c1 <\!\mid\!> c2) \gg\!= k &= (c1 \gg\!= k) <\!\mid\!> (c2 \gg\!= k) \\
c <\!\mid\!> c &= c
\end{aligned}
$$

and the standard laws for reader effects:

$$
\begin{aligned}
\mathit{local}\ f\ \mathit{ask} &= \mathit{ask} \gg\!= \\
&\qquad \lambda x \to \mathit{return}\ (f\ x) \\
\mathit{local}\ \mathit{id}\ c &= c \\
\mathit{local}\ f\ (\mathit{local}\ g\ c) &= \mathit{local}\ (f \circ g)\ c \\
\mathit{local}\ f\ (c1 <\!\mid\!> c2) &= \mathit{local}\ f\ c1 <\!\mid\!> \\
&\qquad \mathit{local}\ f\ c2 \\
\mathit{local}\ f\ (c1 \gg\!= k) &= \mathit{local}\ f\ c1 \gg\!= \\
&\qquad \lambda x \to \mathit{local}\ f\ (k\ x)
\end{aligned}
$$

and writer effects:

$$
\begin{aligned}
\mathit{tell}\ \mathit{mempty} &= \mathit{nil} \\
\mathit{tell}\ o1 \gg \mathit{tell}\ o2 &= \mathit{tell}\ (o1 +\!\!+ o2)
\end{aligned}
$$

and state effects:

$$
\begin{aligned}
\mathit{put}\ s \gg \mathit{put}\ s' &= \mathit{put}\ s' \\
\mathit{put}\ s \gg \mathit{get} &= \mathit{put}\ s \gg \mathit{return}\ s \\
\mathit{get} \gg\!= \mathit{put} &= \mathit{return}\ () \\
\mathit{get} \gg\!= \lambda s \to \mathit{get} \gg\!= k\ s &= \mathit{get} \gg\!= \lambda s \to k\ s\ s
\end{aligned}
$$

We also require that writer and state effects commute with failure. When constructing monad transformers to implement our effect interface, this fixes the order of StateT and WriterT in relationship to MaybeT, namely that they appear higher in the stack.

7.4 Pretty Printing Laws from Monad Laws

Let's start small, from our implementation of $\diamond$ and *nil*:

```
(◇) :: (MonadPretty w ann fmt m) ⇒
       m () → m () → m ()
c1 ◇ c2 = c1 ≫ c2

nil :: (MonadPretty w ann fmt m) ⇒
       m ()
nil = return ()
```

Lemma 7.1. *$m\ ()$, $\diamond$ and nil form a monoid, that is:*

$$
\begin{aligned}
\mathit{nil} \diamond x &= x \\
x \diamond \mathit{nil} &= x \\
(x \diamond y) \diamond z &= x \diamond (y \diamond z)
\end{aligned}
$$

```
chunk (CText s) =
  do tellString s
     addLine s
     checkBreak
 where
  tellString s =
    tell $
      PAtom $
        AChunk (CText s)
  addLine s =
    do format ← askFormat
       modifyLine $
         flip mappend
          [(CText s, format)]
  checkBreak =
    do f ← askFailure
       when (f == CanFail) $
         do wmax ← askMaxWidth
            rmax ← askMaxRibbon
            w ← measureCurLine
            n ← askNesting
            when (n + w > wmax) empty
            when (w > rmax) empty
```

Figure 10. *chunk*, specialized to CText, with named subparts

Proof. Directly from monad left unit, right unit, and associativity laws. □

The implementation of *text* uses *chunk* to insert a text chunk.

```
text :: (MonadPretty w ann fmt m) ⇒ Text → m ()
text t = chunk (CText t)
```

The proofs about *text* rely on various sub-parts of *chunk*. Figure 10 names some of the intermediate computations within *chunk* specialized to CText *s*.

Lemma 7.2. *text is a monoid homomorphism, that is:*

$$text\ (s1 \Vdash s2) = text\ s1 \diamond text\ s2$$

Our proof relies on the fact that failure discards changes in both the state and output, and therefore commutes with state and writer effects.

Proof. First, we unfold the definition of *chunk* in *text*:

```
    text (s1 ⫦ s2)
= {{ unfolding text }}
    chunk (CText (s1 ⫦ s2))
= {{ unfolding chunk }}
    do tellString (s1 ⫦ s2)
       addLine (s1 ⫦ s2)
       checkBreak
= {{ output and line monoid homomorphism }}
    do tellString s1
       tellString s2
       addLine s1
       addLine s2
       checkBreak
= {{ commuting tell and modify effects }}
    do tellString s1
       addLine s1
       tellString s2
       addLine s2
       checkBreak
= {{ idempotent failure effects }}
    do tellString s1
       addLine s1
       checkBreak
       tellString s2
       addLine s2
       checkBreak
= {{ refolding }}
    text s1 ◇ text s2
```

□

Next, our implementation of *nest*:

```
nest :: (MonadPretty w ann fmt m) ⇒ w → m () → m ()
nest i c = localNesting (λi'→i + i') c
```

Lemma 7.3. *If the Num instance for w has 0 as a unit for +, then nest has unit 0 and is distributive through +, ◇ and <|>, that is:*

$$\begin{aligned} nest\ 0\ x &= x \\ nest\ i\ (nest\ i'\ c) &= nest\ (i + i')\ c \\ nest\ i\ (x \diamond y) &= nest\ i\ x \diamond nest\ i\ y \\ nest\ i\ (x <|> y) &= nest\ i\ x <|> nest\ i\ y \end{aligned}$$

Proof. Directly from reader monad unit and distributivity laws. □

Our implementation of Wadler's *group* is a rephrased version of *grouped* from Figure 8.

```
group :: (MonadPretty w ann fmt m) ⇒ m a → m a
group c = ifFlat c $ flatten c <|> c
  where
    flatten = makeFlat ∘ allowFail
```

Lemma 7.4. *group is idempotent and distributes through <|>, that is:*

$$\begin{aligned} group\ (group\ x) &= group\ x \\ group\ (x <|> y) &= group\ x <|> group\ y \end{aligned}$$

Proof. For idempotency:

```
    group (group x)
= {{ unfolding group }}
    ifFlat (group x) $
      flatten (group x) <|>
      group x
= {{ unfolding ifFlat }}
    do f ← askFlat
       if f
       then group x
       else flatten
              (group x) <|>
              group x
= {{ group under askFlat = True }}
```

```
      do f ← askFlat
         if f
         then x
         else flatten x <|>
                group x
= {{ group under askFlat = False }}
      do f ← askFlat
         if f
         then x
         else flatten x <|>
                flatten x <|>
                x
= {{ <|> idempotent }}
      do f ← askFlat
         if f
         then x
         else flatten x <|> x
= {{ refolding group }}
      group x
```

and for distribution:

```
      group (x <|> y)
= {{ unfolding group }}
      do f ← askFlat
         if f
           then x <|> y
           else flatten
                  (x <|> y) <|>
                    x <|>
                    y
= {{ distributivity of flatten }}
      do f ← askFlat
         if f
           then x <|> y
           else flatten x <|>
                  flatten y <|>
                  x <|>
                  y
= {{ nondeterminism distribution }}
      (do f ← askFlat
          if f
            then x
            else flatten x <|> x) <|>
        (do f ← askFlat
            if f
              then y
              else flatten y <|> y)
= {{ refolding }}
      group x <|> group y
```

□

8 Related Work

Pretty printing libraries can be evaluated according to a number of parameters, including the variety of layouts that the document language can describe, the space and time complexity of the string selection algorithm, and the quality of the string selection with respect to efficient use of screen space.

Goldstein [13] provided an algorithm for pretty printing Lisp code that was extensible with custom formatting instructions for built-in operators. This pretty printing system was interactively extensible by users of MacLisp, rather than being a library for expressing a single pretty printer for a fixed language. Goldstein's algorithm performed a great deal of lookahead, and was therefore not suitable for printing large documents.

Oppen [19] described the first efficient general-purpose pretty printing algorithm, in the form of a little language for expressing grouping and literal strings. Presented in an imperative style, Oppen's algorithm requires time linear in the length of the string to be produced and space linear in the width to be used.

A paper by Hughes [15] may be responsible for beginning the Haskell community's love for pretty printing. Hughes provides pretty printing as an exercise in deriving functional programs from their algebraic specification. A variation on his library is used in the Glasgow Haskell Compiler today.

Wadler [25] describes a design for a pretty printing library that is simpler and faster that Hughes's, though there are some layouts that it cannot describe. The Final Pretty Printer implements essentially the same algorithm as Wadler, although noticing this fact requires rewriting it to take less explicit advantage of lazy evaluation. Indeed, after implementing the Final Pretty Printer, the authors noticed that the interplay of *chunk* and *ifFlat* bear a striking resemblance to Ken Friis Larsen's port of Wadler's library to Standard ML.[2]

While Hughes's and Wadler's designs are both elegant and fast enough for many real applications, they do not enjoy the same asymptotic complexity as Oppen's algorithm. Chitil [5, 6] demonstrated that Oppen's algorithm can be implemented in a functional style using lazy dequeues or delimited continuations. Swierstra and Chitil [22] later made this implementation even more clear.

Bernardy [1] managed to provide an implementation of all of Hughes's layouts while providing the very strong guarantee that only the shortest realization of the document as a string will be provided. His implementation takes time linear in the length of the output, using a quality-ranking method to rule out exploration of non-optimal documents.

The expressiveness and performance characteristics of the Final Pretty Printer are roughly similar to Wadler's. In other words, it has worse complexity that Chitil's, and better complexity that Hughes's. In future work, it would be interesting to explore an adaptation of Bernardy's algorithm to a final pretty printer.

[2] Available at the time of writing from https://github.com/kfl/wpp

9 Conclusion

We described the Final Pretty Printer, a pretty printing library that is both highly expressive and extensible. It derives its extensibility from the fact that it is correct in *any* monad, so monad transformers can be used to give it new capabilities, and we demonstrated two widely-applicable extensions. We have implemented a new pretty printer for `cubicaltt`, an implementation of Cubical Type Theory [8], showing that it is practical for real systems, and work is underway to port Idris's interactive environment.

Semantic annotations enable the pretty printer to produce *presentations* that link displayed strings to the meaning that they represent. Presentations have been used in the interactive environment for Idris, enabling interactive error messages, pervasive metadata and documentation, and text decorations that are based on semantics rather than syntax.

By decoupling the measurement of widths from character counts, the Final Pretty Printer enables language developers to properly support almost every natural language in the world, as well as proportional fonts and modern text layout technology. We have demonstrated that it works in both terminal emulators and in Web browsers, two very different environments.

There is no longer any reason to keep our programming environments stuck in the 1970s. The Final Pretty Printer supports today's hardware and today's text rendering technology. And with its built-in extensibility, it also supports tomorrow's.

Acknowledgments

We would like to thank Jon Sterling, Ryan Scott, Dan Friedman, and Sam Tobin-Hochstadt for their feedback on drafts of this paper, and Buddhika Chamith for talking with us about Sinhala on computers.

References

[1] Jean-Philippe Bernardy. 2017. Functional Pearl: a pretty but not greedy printer. In *International Conference on Functional Programming*. To appear.

[2] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *Theorem Provers in Circuit Design*. North-Holland Publishing Co.

[3] Edwin Brady. 2013. Idris, a General Purpose Dependently Typed Programming Language: Design and Implementation. *Journal of Functional Programming* 23, 05 (9 2013).

[4] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. 2009. Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009).

[5] Olaf Chitil. 2001. Pretty Printing with Lazy Dequeues. In *2001 Haskell Workshop*, Ralf Hinze (Ed.). Firenze, Italy. Universiteit Utrecht UU-CS-2001-23. Final Proceedings to appear in ENTCS 59(2).

[6] Olaf Chitil. 2005. Pretty printing with lazy dequeues. *Transactions on Programming Languages and Systems (TOPLAS)* 27, 1 (January 2005).

[7] Eugene Charles Ciccarelli. 1984. *Presentation Based User Interfaces*. Technical Report 794. Massachusetts Institute of Technology Artificial Intelligence Laboratory.

[8] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical Type Theory: a constructive interpretation of the univalence axiom. *ArXiv e-prints* (2016). https://arxiv.org/abs/1611.02108.

[9] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ.

[10] Nils Anders Danielsson. 2013. Correct-by-construction pretty-printing. In *2013 Workshop on Dependently-typed Programming (DTP '13)*. ACM.

[11] Richard P. Gabriel, Nickieben Bourbarki, Matthieu Devin, Patrick Dussud, David Gray, and Harlan B. Sexton. 1990. Foundation for a C++ Programming Environment. In *Proceeding of C++ at Work*.

[12] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] Ira Goldstein. 1973. *Pretty-Printing: Converting List to Linear Structure*. Artificial Intelligence Memo 279. Massachusetts Institute of Technology.

[14] Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Survey* 28, 4es (Dec. 1996).

[15] John Hughes. 1995. The Design of a Pretty-printing Library. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*.

[16] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA.

[17] Scott McKay. 1991. CLIM: The Common Lisp Interface Manager. *Commun. ACM* 34, 9 (Sept. 1991).

[18] Scott McKay, William York, and Michael McMahon. 1989. A Presentation Manager Based on Application Semantics. In *User Interface Software and Technology (UIST '89)*. ACM.

[19] Derek C. Oppen. 1980. Prettyprinting. *ACM Trans. Program. Lang. Syst.* 2, 4 (1980).

[20] Ramana Rao, William M. York, and Dennis Doughty. 1990. A Guided Tour of the Common Lisp Interface Manager. *SIGPLAN Lisp Pointers* IV, 1 (July 1990).

[21] Randall B. Smith, John Maloney, and David Ungar. 1995. The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility. In *Object-oriented Programming Systems, Languages, and Applications (OOPSLA '95)*. ACM, New York, NY, USA.

[22] S. Doaitse Swierstra and Olaf Chitil. 2009. Linear, bounded, functional pretty-printing. *Journal of Functional Programming* 19, 01 (2009).

[23] David Ungar and Randall B. Smith. 1991. SELF: The power of simplicity. *LISP and Symbolic Computation* 4, 3 (1991).

[24] Philip Wadler. 1992. The Essence of Functional Programming. Invited talk. (January 1992). 19th Symposium on Principles of Programming Languages.

[25] Philip Wadler. 2003. A prettier printer. In *The Fun of Programming: A Symposium in Honor of Professor Richard Bird's 60th Birthday*. Oxford.