Format Unraveled

Richard Bonichon, Pierre Weis

Format unraveled

Richard Bonichon[1] & Pierre Weis[2]

1 : CEA LIST, Laboratoire de Sûreté des Logiciels
PC174, 91191 Gif-sur-Yvette Cedex
richard.bonichon@cea.fr
2 : INRIA Paris,
2 rue Simone Iff, CS 42112, 75589 Paris Cedex 12
pierre.weis@inria.fr

Abstract

Pretty-printing can be described as finding a good-looking solution to typeset data according to a set of formatting conventions. Oppen [6] pioneered the field with an algorithmic solution to pretty-printing, using the notions of *boxes* and *break hints*. The `Format` module is a direct descendant of this work: it is unfortunately often misunderstood or even misused. The first goal of this article is to enhance the available documentation about `Format` by explaining its basic and advanced features but also its relationship and differences with Oppen's seminal work. The second goal is to investigate the links that `Format` has with the document-based pretty-printing tradition fostered by the lazy programming community [3, 4, 9, 10].

1. Introduction

Reading and printing data are usual parts of day-to-day programming. As a witness to the truth of this statement, OCaml has two modules concerned with reading data (`Pervasives`, `Scanf`) and even more — three — with printing (`Pervasives`, `Printf`, `Format`).

Usually, we want to obtain a certain regularity in this output, to have it *formatted*. A formatted output can be made to look more or less pretty. The definition of prettiness as a value is a rather philosophical matter [7]: nonetheless this is the goal of *pretty-printing*. That is, a pretty-printing module should have functionalities in order to help output structured data in a good-looking way.

In OCaml, this can be achieved via the standard library with one of the two modules dedicated to the formatted output of data. On the one hand, there is the `Printf` module, which can be roughly described as an extended look-alike of the C family of printf functions. On the other hand, there is the `Format` module. At first sight, it seems rather similar to a fusion of `Printf` and `Pervasives`. But it comes with extra advanced capabilities which are often misused or misunderstood. The first goal of this article is to document extensively what the `Format` module offers, how it works (and why it works that way), and how to use it.

Other programming languages, such as Python or Java, often have a format function or class. However, these are usually not akin to OCaml's `Format` module, and more like `Printf`. Indeed, their names probably come from the fact that they are based upon *format strings*.

Pretty-printing a la `Format` seems to be more common in functional languages. The `Format` module is indeed based on early work done by Derek Oppen [6]. Haskell has for example received a good amount of attention through, first, the works of Hughes [3] and Wadler [10]. Their work relies on the exploration of algebraic properties of Oppen-like pretty-printing and traditionally defines pretty-printing combinators.

This line of work reaches the same level of efficiency as Oppen's algorithm only with Swierstra and Chitil [9]: their functional pearl describes a combinator-based pretty-printing algorithm that have the same space (w.r.t to the line width) and time (w.r.t to the length of the stream) complexities as Oppen's while retaining a lazy functional flavor. Later, Kiselyov, Peyton-Jones and Sabry [4] show how to program an elegant incremental pretty-printer using yield.

The contributions of this paper are the following:

- it revisits, complements and extends the standard documentation available for `Format`;
- it explains the differences between Oppen's original algorithm and what `Format` offers;
- it discusses how `Format` differs from document-based pretty-printers advocated by the algebraic pretty-printing tradition and investigates how it can be made similar to them.

2. A brief history of `Format`: a quest for abstraction

The `Format` module has a long history of development, starting from the early implementation of Oppen's algorithm in Caml circa 1985 to the full-fledged module we describe in this article.

The first implementation started for the compiler internal needs to print error messages and computed values. So far so good, the pretty-printer was able to decently display types and values on the terminal, when Caml was only available as an interactive system. The first step toward abstraction was to encapsulate the pretty-printer as a module and export it to user land.

Then, the problem arose not to mix compiler messages and warnings with the output of the user's program. It was time to add separate printing functions for stderr, stdout and general output channels. The idea of abstracting the low-level output device from the pretty-printing engine was born.

To allow parallel pretty-printing to several files and output channels, the entire pretty-printer had to be abstracted. Here comes the `Format.formatter` data structure: a *formatter* encapsulates a complete pretty-printing engine with all its state and specific parameters into a value that can be manipulated in programs.

Around 1990, Caml-Light added basic format strings to properly typecheck the `printf` function and provide a safe use of format strings. The introduction of a `Format` specific version of the `printf` family of functions gave rise to the addition of specifications for boxes and break hints management directly into format strings. Polymorphic printing with explicit formatter arguments was then made available via conversion `%a` (see Section 5).

The quest for abstraction went on with *semantic tags* (Section 6), printing with continuations, and recently output abstraction with *symbolic printing*[1]. The future is rich with further endeavors (see Section 10).

3. `Format` basics

`Format` can write on anything that can receive characters, such as strings, buffers, channels or streams. In this section, for sake of simplicity, we focus on basic primitives writing on the terminal (stdout). `Format`'s basic primitives can be divided into two sets: primitives to print elementary values and primitives for indentation and splitting lines (Sections 3.1 and 3.2).

Printing elementary values in `Format` is similar to printing those values with the fundamental `Pervasives` module. One can print characters, strings, integers, floats and booleans with `print_char`, `print_string`, `print_int`, `print_float` and `print_bool`. The `print_newline` primitive in `Format`

[1] `https://github.com/ocaml/ocaml/pull/615`

prints a newline character as the corresponding `Pervasives` function, buts its impact on the pretty-printing engine is major and should not be underestimated (see Section 7).

3.1. Break hints

A *break hint* is an explicit annotation to tell the pretty-printing engine where it can split the line. A break hint also indicates the amount of spaces to add to the current indentation when splitting the line.

Break hints for the pretty-printing engine can be given with the `print_space`, `print_cut` and `print_break` functions.

The first function outputs a *space* break hint: it outputs a typographical space if there is no need to split the line or it splits the line according to the box discipline without adding indentation.

The second one outputs a *cut* break hint: it does nothing if there is no need to split the line or it splits the line according to the box discipline (no indentation added).

The last one outputs a *full* break hint: it has two parameters nspaces and offset. It outputs nspaces typographical spaces if there is no need to split the line or it splits the line according to the box discipline, adding offset spaces to the current indentation value. Those integer parameters can be negative: a negative nspaces is treated as 0, while a negative offset reduces the indentation of the next line. Note that space and cut break hints are convenient shortcuts for specific full break hints.

3.2. Boxes

A *pretty-printing box*, or simply a *box*, is the fundamental device which delimits a region with a coherent discipline of line-splitting and indentation.

There are five line-splitting disciplines corresponding to five types of boxes, with different effects on the output. Those types are h, v, hv, hov and b. h stands for horizontal, v for vertical, hv for horizontal/vertical, hov for horizontal-or-vertical and b for basic.

Each box type is respectively opened with `open_hbox: unit -> unit`, `open_vbox: int -> unit`, `open_hvbox: int -> unit`, `open_hovbox: int -> unit`, `open_box: int -> unit`. When lines can be split, boxes have an extra indentation argument that specifies the amount of extra spaces added to the current indentation of the block when splitting lines. Let us now detail these boxes (Figure 1 shows a comparative look at their behaviors).

```
let pp_int_list open_box l =
  let rec pp = function
    | [] -> ()
    | [x] -> print_int x
    | x :: xs ->
      print_int x; print_space (); pp xs in
  open_box 0; pp l; close_box ()
```

```
let pp_int_list_h   =
  pp_int_list (fun _ -> open_hbox ())
and pp_int_list_v   = pp_int_list open_vbox
and pp_int_list_hv  = pp_int_list open_hvbox
and pp_int_list_hov = pp_int_list open_hovbox
and pp_int_list_b   = pp_int_list open_box;;
set_margin 8
```

```
1 2 3 4 5
```

(a) *h-box*

```
1
2
3
4
5
```

(b) *v-box*

```
1 2 3 4
5
```

(c) *hov-box*

```
1 2 3 4
5
```

(d) *b-box*

```
1
2
3
4
5
```

(e) *hv-box* (non-fitting)

```
1 2 3 4 5
```

(f) *hv-box* (fitting)

Figure 1: Comparing box splitting discipline (margin 8, except margin 10 for 1f)

Horizontal boxes A *horizontal box* or *h-box* groups contents to be printed on a single line, thus hiding the column limits of the pretty-printing engine. One idiosyncrasy to *h-boxes*: if the size of a horizontal box is bigger than the margin size left on the output device, its whole contents is printed on the next line.

Vertical boxes A *vertical box* or *v-box* groups contents whose elements must each be printed on a separate line.

Horizontal/vertical boxes A *horizontal/vertical box* or *hv-box* has two mutually exclusive behaviors: if the box fits on a single line, the box is said *fitting* and behaves as a horizontal box; otherwise, the box is said *non-fitting* and behaves as a vertical box.

Horizontal-or-vertical boxes A *horizontal-or-vertical box* or *hov-box* is a compacting box: it outputs its contents on the same line while there is enough room left on the line. Then, the next break hint splits the line and the output goes on. A text output in a horizontal-or-vertical box with all spaces used as break hints is similar to left-justified paragraph in a text processor.

Basic boxes A *basic box* or *b-box* is a compacting box similar to the horizontal-or-vertical box with a different way to handle break hints: if splitting the line reduces the current indentation, a break hint splits the line, even if there is still enough room left on the current line.

Comparing compacting boxes: *b-box* versus *hov-box* Figure 1 shows that *hov-boxes* and *b-boxes* behave the same in simple cases. However, printing complex material with nested boxes shows up the difference.

Figure 2 prints the same list of integers with the same pretty-printing function as Figure 1. In addition, Figure 2 uses a global compacting box to properly pretty-print the list between brackets. Both parts of Figure 2 run the exact same code except for the enclosing box: a *hov-box* for Figure 2a and a *b-box* for Figure 2b. The right margin is set to 8, thus *there is enough room* to print the closing bracket on the second line. In the case of the *hov-box*, there is no need to split the line at the *cut* break hint before the closing bracket. In the *b-box* case, the *cut* break hint splits the line, since *splitting the line reduces the current indentation*, and the closing bracket is displayed on a new line. This behavior aligns opening and closing delimiters, thus enforcing the list structure.

This small example is too simple to show the true benefit of the *b-box* behavior. A more complex example, for instance printing a tuple of lists of records would be more telling. If such a value is printed within a *hov-box*, all the closing parentheses, brackets and braces appears at the end of line, possibly all in a row on the last line: the *hov-box* minimizes the number of lines of the output. This is less readable than using a *b-box*, which may add extra lines to emphasize the box structure, printing each closing character on a new line, properly indented with its opening sibling. In short, the *b-box* visually enhances the structure of the value; that is why a *b-box* is also known as a *structural* compacting box.

```
open_hovbox 0;
 print_string "[";
 pp_int_list_hov l;
 print_cut ();
 print_string "]";
close_box ();;
```

```
[1 2 3
 4 5 ]
```

(a) Printing a list inside a *hov-box*

```
open_box 0;
 print_string "[";
 pp_int_list_hov l;
 print_cut ();
 print_string "]";
close_box ();;
```

```
[1 2 3
 4 5
]
```

(b) Printing a list inside a *b-box*

Figure 2: Behavior comparisons: *b-box* vs. *hov-box* (margin 8)

3.3. Remarks

Before concluding this section, we would like to provide a bit of context by discussing the reasons why `Format`'s primitives and behaviors are profoundly different to common text processors and to share some perspective with respect to what has been added thus far to Oppen's algorithm.

3.3.1. Pretty-printing versus text processing

The break treatment in `Format` is the reverse of usual text processing software where a normal space is breakable and you need indicate hard (non-breaking) spaces. This salient difference is on purpose and due to the somewhat opposite design and goals of text processing and pretty-printing software.

In text processing, the input is structured via paragraphs, sections and subsections. Paragraphs are free flowing streams of words separated by spaces and punctuation signs. The job of the text processor is to respect and emphasize section markers and properly split paragraphs; clearly, spaces in paragraph should default to breakable, while spaces in section titles are certainly unbreakable. The adoption of such spacing conventions leads to almost no breakable annotations for spaces.

In text processing, all breakable spaces behave the same: they all output a typographical space or open a new line starting at margin. Furthermore, splitting a paragraph after a word or the next is not a dramatic decision: a document typeset without following best practice remains perfectly readable and understandable.

By contrast, in pretty-printing, the primary input is a computed value of some structured data. The job of the pretty-printer is to help the programmer to properly split the lines to respect and emphasize the internal structure of the value. Here, splitting a line and indenting the next one is of utmost importance to highlight this internal structure. Hence, the pretty-printer provides ways to carefully fix the indentation; in particular, `Format` boxes and break hints carry an argument to indicate the indentation of new lines.

In pretty-printing, break hints fix the indentation of lines, so each break is specific. Furthermore, splitting a line at this break hint or at the next one is a dramatic decision that could wreck havoc the final document to the point that it becomes unreadable and difficult or even impossible to understand. This is precisely the case for some programming languages where indentation is significant such as Python and Haskell.

As a final fundamental contrasting difference, the text in text processing is mostly hand-written and contains hand-written spaces, when the text in pretty-printing is mostly machine-generated by hand-written programs that compose small pieces of text separated by machine-generated break hints.

3.3.2. Oppen's algorithm

Oppen's algorithm [6] is at the core of `Format` pretty-printing engine and also the basis for algebraic studies of the lazy community. This section sums up its main components and insights.

In his article, Oppen introduces the notions of box and break hint (called a *blanks*). Inside a box, blanks can be *consistent*, causing a `Format`*hv-box* or *inconsistent*, yielding a `Format`*hov-box*. Each blank has a length and an offset, just as in `Format`.

The algorithm is based on the interplay between two functions: print and scan. The latter represents the stream to be pretty-printed. The former effectively prints the material: a string is always printed; if a box is open, its indentation is pushed on a stack; if one is closed, it is popped; if a blank is received, it is printed if it fits on the line otherwise the line is split and indented according to this blank and the current box (on the top of the stack). The latter appends tokens to a buffer: to each string and *openbox* token, it also associates its length; to each blank, it associates the length of the blank plus the length of the next block, in order to check if it can print the coming block.

The core of the algorithm is very similar to `Format` internals. `Format` retains the same linear complexity as Oppen's proposal. Furthermore, `Format` adds the two sub-components of the *hv-box*, the *h-box* and the *v-box*, as well as the *b-box*. It also supports fully typed format strings, semantic tags and, last but not least, abstraction.

4. Format strings

In OCaml, there exists a basic notion of *format string value* with a corresponding *format string type*. A *format string value* is a concise way of specifying a sequence of *value arguments*, in particular the *type* and *shape* of each argument of the sequence. Since OCaml language constructions have to be statically typechecked, arguments of input/output procedures should be specified so that their types can be verified. *Format string values* are the natural polymorphic way to specify all the arguments of advanced input/output functions: indeed, *format string values* specify any sequence of values to be read using module `Scanf` or printed using modules `Format` or `Printf`.

4.1. Syntax of format strings

The syntax of *format strings* is identical to the syntax of OCaml basic strings, namely a sequence of characters between double quotes. However, sequence of characters inside *format strings* must obey a specific and constraint syntax to describe types and shapes of arguments.

Following the C tradition, argument specifications are introduced by special marker `%`, followed by a letter giving the type of the argument. For instance, `%i` indicates an integer argument and specifies type `int` for this argument. Still following the C tradition, argument specifications are called *conversions*. OCaml *format strings* support specific conversions for basic types, such as `string` with `%s`, `float` with `%f`, `bool` with `%b`, `char` with `%c`, and so on.

Apart from types, argument shapes may be specified via several means. One can *indicate an alternate conversion*: for instance all conversions `%d`, `%x`, `%X`, and `%o` specify an integer argument, but each of those conversions fixes a different notation for the integer. Indeed, `%d` prints (or reads) decimal digits, `%x` or `%X` hexadecimal digits, and `%o` octal digits. Similarly, both `%s` and `%S` specify a string value, but conversion `%S` specify a string delimited with double quotes and using the OCaml lexical conventions to escape characters. Also, one can add optional size and precision specifications by extra characters after the conversion marker (for instance `%4.12g`), as well as padding and alignment annotations, which are absent from the basic functionalities described in Section 3.

Format strings can also contain material unrelated to argument specifications: the *formatting indications* do not specify the type or shape of arguments but the *presentation* of arguments. The presentation indicates *how* arguments should appear in a document (i.e. in a sequence of characters). That is, a *formatting indication* states how to display an argument when printing, or how to read an argument when scanning. In *format strings*, such a *formatting indication* is introduced by the special marker `'@'`, followed by a sequence of letters specifying the indication.

Note that *formatting indications* do not interfere with the typing of *format strings*. Interpretation of *formatting indications* may also be module specific: some *formatting indications* for reading are not meaningful for printing and vice versa.

There is a complete set of *formatting indications* to drive the `Format` pretty-printing engine: opening and closing *formatting boxes*, emitting break hints, even flushing the pretty-printing engine to terminate a pretty-printing routine.

For instance `"@["` opens a box and `"@]"` closes the last opened box. Similarly, *formatting indication* `"@ "` emits a space break hint and `"@,"` emits a cut break hint. When a *formatting indication* needs an argument, it has to be enclosed between characters `'<'` and `'>'`; for instance, adding a box kind argument to the box opening *formatting indication* gives `"@[<h>"`, `"@[<v>"`, `"@[<hv>"`, `"@[<hov>"`, and `"@[<b>"` to open the corresponding boxes (defined in Section 3.2). Figure 3 shows how to reproduce Figure 1 with boxes in *format strings*. If another additional argument is necessary, simply add it after a space: `"@[<v 2>"` opens a vertical box with 2 as indentation increment. Similarly, a full break hint is introduced by `"@;"` and needs two integer arguments: it is written as `"@;<1 2>"`.

The last set of characters in *format strings* are *plain characters*, that is any character not preceded

```
open Format

let pp_format box_type ppf l =
  let pp_list ppf =
    List.iter (fprintf ppf "%d@ ") in
  fprintf ppf "@[<%s>%a@]" box_type pp_list l

let pp_format_h   = pp_format "h"
and pp_format_v   = pp_format "v"
and pp_format_hv  = pp_format "hv"
and pp_format_hov = pp_format "hov"
and pp_format_b   = pp_format "b"
```

Figure 3: Box type test with format strings (results as in Figure 1)

by the *conversion marker* `%` nor by the *formatting indication marker* `@`. This is regular text included in a *format string* to be output or read as verbatim material. Note that markers are considered plain characters if preceded by a `%` character; write `%%` or `%@` to obtain a plain `%` or a plain `@` character.

Do not be confused by the specific usage of *format strings*: they are *first-class citizen* of the language. Hence, a *format string* can be returned as a result, passed as an argument or manipulated as any other value. For instance, the predefined infix operation `^^` implements the concatenation of *format strings*: `fmt1 ^^ fmt2` is equivalent to *format string* `fmt1` followed by *format string* `fmt2`. The function `Pervasives.string_of_format` gives the string representation of any *format string*. Conversely, `Pervasives.format_of_string` returns the *format string* value corresponding to a string with known characters (a *string literal*). To convert to *format string*, a statically unknown string, for example a string read from a file, use `Scanf.format_from_string` (see Section 4.3).

Caveat: pattern matching for *format strings* is not yet available but comparing their string representations may help.

4.2. Typechecking format strings

The typing of *format strings* is specific, complex, and highly polymorphic. Indeed, the general type constructor able to accommodate all *format strings* peculiarities needs 6 different type variables: this holds the record for the most polymorphic datatype of the entire OCaml library.

Format strings have a general and highly polymorphic type `('a, 'b, 'c, 'd, 'e, 'f) format6`. Let's give more meaningful names to those type variables renaming them respectively as `'functional_type`, `'low_level_device`, `'poly_printer_result`, `'poly_reader_functional_type`, `'poly_reader_result`, `'result_type`.

`'b` (`'low_level_device`) is the type of the low-level device for the format string, an input device for scanf-like functions and an output device for printf-like functions

`'f` (`'result_type`) is the result type of the format string, it is the result type of the receiver for scanf-like functions and the result type of printf-like functions

`'a` (`'functional_type`) is `'argument_sequence -> 'result_type`, where `'argument_sequence` is the type of the sequence of arguments to print or of values to read. For the `Scanf` family of functions `'functional_type` is also the type of the receiver function.

`'c` (`'poly_printer_result`) is the result type of polymorphic pretty-printers required by `%a` conversions in the format string (hence a polymorphic pretty-printer printing values of type `'t` has type `'low_level_device -> 't -> 'poly_printer_result`). Conversion `%a` is detailed in Section 5.

`'d` (`'poly_reader_functional_type`) is `'poly_reader_sequence -> 'poly_reader_result`, where `'poly_reader_sequence` is the type of the sequence of polymorphic readers required by all the

`%r` conversions in the format string, and `'e` is the result type of `'poly_reader_functional_type` (hence a polymorphic reader reading values of type `'t` has type `'low_level_device -> 't`).

4.3. Typing primitive functions on *format strings*

The function `string_of_format` maps any *format string* to its corresponding string representation. Hence, its type scheme is naturally: `('a, 'b, 'c, 'd, 'e, 'f) format6 -> string`.

On the other hand, the type of `format_of_string` is

```
('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6
```

This is surprising in more than one way. First, because the input type of the function is *not* `string`! Second, because that type is an instance of the type scheme of the identity function (namely, the type of identity restricted to *format strings*).

So, in the first place, how can `format_of_string` be applied to a value of type `string` when its source type is `_ format6` ? However, the function indeed converts a string to a *format string*, as in

```
# format_of_string "%d";;
- : (int -> '_a, '_b, '_c, '_d, '_d, '_a) format6 = "%d"
```

There is some black magic at work here. It indeed lies in the typechecking of string *constants*. In presence of a string *constant* expression, the typechecker follows a pragmatic rule: if the expression is expected to be a *format string*, then its contents is analyzed to discover its `format6` type; otherwise, it gets type `string`. For instance:

```
# ("%d" : _ format6);;
- : (int -> 'a, 'b, 'c, 'd, 'd, 'a) format6 = "%d"
```

On the other hand, if the string constant is bound to identifier `s`, then it gets type `string` and cannot be applied to `format_of_string` anymore:

```
# let s = "%d" in format_of_string s;;
Error:  This expression has type string but an expression was expected of type
  ('a, 'b, 'c, 'd, 'e, 'f) format6
```

The documentation clearly states it: `format_of_string` converts a string *literal* to a *format string*. In fact, `format_of_string` simply checks that a string constant is a valid *format string*.

If you need to convert any string, not only a literal string, you need `Scanf.format_from_string` that can read any string and convert it using a *format string* pattern. `Scanf.format_from_string` has type `string -> ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6`.

The first `string` argument is simply the string to be converted, but the second argument is more intriguing: it is the *model* of the expected *format string* result: a static witness for the type of the expected *format string* result.

```
# let s = "Price = %.2g" in Scanf.format_from_string s "%f";;
- : (float -> '_a, '_b, '_c, '_d, '_d, '_a) format6 = "Price = %.2g")
```

`Scanf.format_from_string` indeed verifies that the given string can be assigned the type of the second argument *format string* pattern.

5. Polymorphic printing

`Format`'s killer feature trio is `fprintf`, `formatter`, `%a`: this is the way to polymorphic and compositional pretty-printing.

A *formatter* is the abstraction of a complete pretty-printing engine that can be specialized to various tasks: the low-level output device, the parameters for margins, the treatment of various semantic aspects of the pretty-printing engine can all be encapsulated into a *formatter*. For instance, use `formatter_of_out_channel` to get a formatter that outputs to a given `out_channel`, or `formatter_of_buffer` to get a formatter that outputs to an extensible string buffer. A routine with an explicit `formatter` argument is completely generic with respect to the pretty-printing engine and is called a *pretty-printer*. According to its *formatter* argument, the routine can write to *any* low-level output device; more importantly, it can behave according to any high-level pretty-printing abstraction that can be defined as a `formatter`.

The function `Format.fprintf` is such a generic pretty-printer and in fact the most general in `Format`. `fprintf` takes a `Format.formatter` as first argument (in the name `fprintf`, f stands for formatter). This way, `fprintf` subsumes the entire `printf` family: choosing the formatter argument turns `fprintf` to a specific function. For instance, `printf`, `eprintf`, `sprintf` are equivalent to using `Format.fprintf` with `std_formatter`, `err_formatter`, `str_formatter`.

The polymorphic conversion specification `%a` is a specific addition to OCaml *format strings*. Intuitively, `%a` means "use the following function to convert the next argument". So in fact `%a` specifies two arguments, a function `f` and a value `x` so that `f` can print value `x`. More precisely, `f` must print `x` on the *low-level device* specified by the *format string* that includes the `%a` conversion. Hence, if `fmt` has type `(_, 'low_level_device, _) format6` and `x` has type `'t`, then `f` must have type `'low_level_device -> 't -> ...`. In short, `f` must be a pretty-printer.

Conversion `%a` is particularly noteworthy because, as the type indicates, the conversion is *polymorphic*. Furthermore, since `%a` also abstracts a function, it allows composition of pretty-printers. In short, `%a` is the truly functional *format string* conversion!

Conversion `%a` and `fprintf` at work To illustrate pretty-printer composition, we write a pretty-printer for a simple expression algebraic datatype, then a polymorphic pretty-printer for pairs of values.

```
let pp_int ppf = fprintf ppf "%d"

let pp_pair pp_x pp_y ppf (x, y) =
  fprintf ppf "@[(%a,@ %a)@]" pp_x x pp_y y

let pp_int_pair = pp_pair pp_int pp_int
```

```
let rec pp_expr ppf = function
  | Int n -> fprintf ppf "%i" n
  | Add (e1, e2) ->
    fprintf ppf "(%a@ +@ %a)"
      pp_expression e1 pp_expression e2

and pp_expression ppf =
  fprintf ppf "@[%a@]" pp_expr
```

Figure 4: `Format.fprintf` at work

The pretty-printer for simple integer expressions with addition uses the format string `"(%a + %a)"` to write additive expressions. The version given in Figure 4 adds break hints and ensure proper boxing through two mutually recursive pretty-printers. As we can see, the composition of pretty-printers via the `%a` conversion has one peculiarity: *the formatter argument is implicitly applied* to the pretty-printing function argument.

The polymorphic pair pretty-printer uses two `%a` conversions to print each element of the pair; hence, it needs abstract two pretty-printers and a formatter. Then it uses a *format string* like `"(%a, %a)"`. Adding formatting indications to the *format string*, we get the `pp_pair` function of Figure 4.

To define a pretty-printer for specific pairs, simply follow the usual functional programming way and apply `pp_pair` to two pretty-printers as in `pp_int_pair`.

6. Semantic tags

`Format` offers another extension to Oppen's original proposal. It has the ability to interpret specific pretty-printing hints called semantic tags. In format strings, a tagged section is delimited by `"@{<t> ... @}"` for tag `t`. The interpretation of those tags is purely user-driven as the programmer must supply appropriate `open_tag` and `close_tag` functions. These come in two flavors: marking and printing when tags are respectively opened and closed.

Tag printing functions are intended to emit formatting instructions (open a box, put a break, etc.) while tag marking functions simply emit a 0-length string marker associated to the tag. Basic use of tag marking functions is for example to print opening and closing markers in HTML. As tag markers are considered of length 0, they do not interfere with line splitting or indentation. Also note the order of invocation of tag handling functions: when a tag is opened, `print_open_tag` is called first then `open_mark_tag`; when a tag is closed, `close_mark_tag` is called first, then `print_close_tag`.

We illustrate using semantic tags with two examples. The first example uses tags to optionally enable color printing for terminal outputs. The second provides two different outputs from the same tagged content. Both cases are handled almost seamlessly with semantic tags.

Colors The first example consists in coloring the output, for example for a logging module. Tags can be turned on or off depending on the output device. The interpretation of how to color the output could even be device dependent. This example has two modes: when the output is done on a terminal, tags emit ANSI color escape sequences, otherwise they are left uninterpreted. The latter is better if the output formatter is a device where color escape sequences have no special meaning.

In this example (see Figure 5), we restrict ourselves to three foreground colors: yellow, purple and cyan, whose corresponding escape sequences[2] are 33, 35 and 36. All attributes are off by default (hence the additional 0), except for yellow which is bold (represented by the value 1). The `mark_close_tag` function always emits a "reset to default" sequence.

Different outputs for the same tags Tags provide means to have different concrete outputs for the same initial data. In this case, tags can be seen as ways to embed simple node annotations into the output. One could annotate the pretty-printer of any datatypes with simple tags and process these tags differently according to the desired output (you could thus "serialize" a type through tags).

The example has two simple outputs, one in HTML and one in Emacs org [2] format for the same initial set of tags. The code is shown in Figure 6. Note that the `print_open_tag` and `print_close_tag` functions do not have a formatter argument: they would always print to `Format.std_formatter` if left alone. Thus, it is necessary to bind them to the proper formatter whenever using them.

For HTML, we want the closing `</ul>` tags to be indented the same as its opening companion. This is the primary use of a basic box. Then, we want all `<li>` list items to be vertically aligned inside the list. This desired behavior mixes boxes and printing, thus it can only be defined in tag printing functions and not in tag marking functions. Note that we print `<ul>` tags as 0-length items. In effect, this mimics what marking functions do.

In org, lists are simple vertically aligned paragraphs, preceded by a - sign. The usual notion of paragraph is handled by a *hov-box* in the `print_open_tag` and `print_close_tag` functions.

[2] `http://ascii-table.com/ansi-escape-sequences.php`

```
let str_to_esc_seq color_name =
  match String.lowercase color_name with
  | "cyan"   -> Some "0;36"
  | "purple" -> Some "0;35"
  | "yellow" -> Some "1;33"
  | _        -> None

let color_tag_funs =
  { mark_open_tag = (fun tag_string ->
        match str_to_esc_seq tag_string with
        | None      -> ""
        | Some eseq -> sprintf "\027[%sm" eseq);
    mark_close_tag  = (fun _ -> "\027[0m");
    print_open_tag  = (fun _ -> ());
    print_close_tag = (fun _ -> ()); }
```

```
let pp_colorized ppf fmt =
  pp_set_formatter_tag_functions ppf color_tag_funs;
  let mark_tags = pp_get_mark_tags ppf () in
  pp_set_mark_tags ppf true;
  kfprintf (fun ppf -> pp_set_mark_tags ppf mark_tags)
    ppf fmt ;;

pp_colorized std_formatter
  "@[<v 0>Default@ \
   @{<cyan>Cyan@}@ \
   @{<yellow>Bold Yellow@}@ \
   @{<purple>Purple@}@}@ \
   @{<uninterpreted>Default@}@]@."
```

Figure 5: Colored terminal output with semantic tags

```
open Format

let fmt = format_of_string
  "@[<v 0>\
   @{<p>This paragraph precedes a list:@}@ \
   @{<ul>\
     @{<li>This@ first@ item@ might@ be@ too long@}\
     @{<li>Second item@}\
   @}\
   @]@.\
  "
```

```
let dedicated_pp tag_functions ppf fmt =
  pp_set_margin ppf 35;
  pp_set_formatter_tag_functions ppf tag_functions;
  let mark_tags = pp_get_mark_tags ppf ()
  and print_tags = pp_get_print_tags ppf () in
  pp_set_mark_tags ppf true;
  pp_set_print_tags ppf true;
  kfprintf (fun ppf ->
      pp_set_mark_tags ppf mark_tags;
      pp_set_print_tags ppf print_tags)
    ppf fmt
```

```
let html_tag_functions ppf =
  let mark_open_tag s =
    if s <> "ul" then "<" ^ s ^ ">" else ""
  and print_open_tag  = function
    | "ul" -> fprintf ppf "@[<b>@<0>%s@[<v 2>" "<ul>"
    | "li" -> fprintf ppf "@ @[<hov 0>"
    | "p" -> fprintf ppf "@[<hov 0>"
    | _ -> ()
  and print_close_tag  = function
    | "ul" -> fprintf ppf "@]@,@<0>%s@]" "</ul>"
    | "li" -> fprintf ppf "@]"
    | "p" -> fprintf ppf "@]"
    | _ -> ()
  and mark_close_tag s =
    if s <> "ul" then "</" ^ s ^ ">" else ""
  in { mark_open_tag;  mark_close_tag;
       print_open_tag; print_close_tag; }

let pp_html ppf =
  dedicated_pp (html_tag_functions ppf) ppf ;;
pp_html std_formatter fmt
```

```
let org_tag_functions ppf =
  let mark_open_tag _ = ""
  and print_open_tag  = function
    | "ul" -> fprintf ppf "@[<v>"
    | "li" -> fprintf ppf "- @[<hov>"
    | _ -> ()
  and print_close_tag  = function
    | "ul" -> fprintf ppf "@]"
    | "li" -> fprintf ppf "@]@ "
    | _ -> ()
  and mark_close_tag _ = ""
  in { mark_open_tag;  mark_close_tag;
       print_open_tag; print_close_tag; }

let pp_org ppf =
  dedicated_pp (org_tag_functions ppf) ppf ;;
pp_org std_formatter fmt
```

```
<p>This paragraph precedes a list:</p>
<ul>
  <li>This first item might be too long</li>
  <li>Second item</li>
</ul>
```

```
This paragraph precedes a list:
- This first item might be
  too long
- Second item
```

Figure 6: Tag interpretation for different outputs

7. Guidelines for using Format

Proper use of Format requires a certain discipline to maximize its help. Here are some guidelines.

Guideline 1 (Boxing rules)**.** Before using Format, *thou shalt know thy boxes.* In particular:

1. If you do not open a box, there is no guarantee and no semantics.
2. When the pretty-printer is reset, it empties all its stacks and queues **and**, as of today, open a *b-box* with offset zero. This has changed in the past and could change again any release.
3. So, a box is open by default. But, as you cannot assume which one, you shall always open one.

Guideline 2. *Format helps those who help Format.* Do not hesitate to add break hints or open new boxes. This helps to avoid various symptoms such as: way too long lines or contents spread on many small lines vertically aligned at the right margin.

Remember: the cost of opening boxes and adding break hints is dwarfed by the cost of outputting the content.

Guideline 3 (Flushing discipline)**.** It is mandatory to flush the pretty-printing engine *at the end of pretty-print*, to print all the material waiting for good rendering in the pretty-printing engine data structures.

You shall *not* flush the pretty-printing engine *at random*, either using `"@."` (`print_newline`) or `"@?"` (`print_flush`), because flushing automatically *closes all boxes and tags.* This breaks the box splitting discipline.

Guideline 4 (Newline)**.** Formatting indications for newline `"@\n "`, or flush and newline `"@."` are delicate to use.

Adding a newline which is not computed by the pretty-printing engine is risky at best for it breaks the box splitting discipline.

If you need to split lines, simply open a v-box and output normal break hints: inside a v-box, each break hint will print a newline as desired, but all open boxes will stay active and the document rendering will continue normally. As an extra benefit, inside a v-box line splitting occurs without low-level device flush, thus usually improving efficiency.

Guideline 5 (Use `fprintf` and `%a`)**.** Function `fprintf` gets a `Format.formatter` first argument. Via `%a` conversions, `fprintf` can compose pretty-printers in a generic and natural way.

Guideline 6 (Abstract the formatter)**.** To make your routines generic and compatible with `%a`, promote them to pretty-printers by adding an explicit `Format.formatter` argument.

8. Document generation with Format

There are mainly two traditions when it comes to pretty-printing. In the functional world, a document-based approach has garnered much attention. The other tradition comes directly from Oppen's seminal article and has lead to the Format module in OCaml. This section discusses those two different approaches. Spoiler: this has a lot to do with the fundamental lazy/strict differences. We will also show how one can extend Format to create a document.

Document-based pretty-printing has been championed by Hughes [3] and Wadler [10], promoting thereby the ability to work at an algebraic level. In this setting, a document can be abstracted as either a string (`Text`), a potential line break with indentation (`Line`), the concatenation of two documents (`Concat`), or a group, that is a unit with line breaks interpreted consistently. That is why a group is translated to a Format hv-box. The `document` type is defined in Figure 7.

In a lazy setting (call by name/need) values may be suspensions that are not yet been completely computed, but will be computed as much as desired "on demand". For instance, appending lists can cost almost nothing since elements of the resulting list will be constructed and consumed as necessary by the function using the result (call by need features a kind of "pipeline effect for free"!). In a strict setting (call by value), data must be completely built before usage: in the case of list concatenation, it means that the entire resulting list is built before its first element is made available.

This could partly explain why building a complete document before printing is in some sense a conceptual notion in a lazy setting: the parts of the document will be build on demand, while printing the document. No extraneous data structure is built before printing, the values to be printed are computed and used to drive the pretty-printing routines. In a strict setting, these values are completely computed and built anyway, so there is no extra cost in pretty-printing them.

Hence, in a strict setting, building documents could be much more expensive, since the final document data structure is entirely built before printing starts. Using modern computers, document construction could be fast enough to be tolerable or amount to a fraction of the total cost of pretty-printing. Actually, in the blog post announcing `Pprint`[3], an OCaml library providing combinators for building and printing documents, Pottier similarly notes:

> One limitation of the library is that the document must be entirely built in memory before it is printed. So far, we have used the library in small-to medium-scale applications, and this has not been a problem. In principle, one could work around this limitation by adding a new document constructor whose argument is a suspended document computation.

As said before, document building is driven by values. In `Format`, values drive the pretty-printing engine without the need of a document. In a way, `Format`'s pretty-printing engine simply avoids the construction of documents or uses a virtual construction of the document that it prints before even building it! Building a document instead of pretty-printing could simply be a debugging option of the pretty-printing engine: it could be useful to match the semantics of pretty-printers by looking their pretty-printing meaning, instead of painfully guessing from the pretty-printing engine output.

This suggests to add to the pretty-printing engine a *formatter* that would build a document instead of printing its virtual representation. Such an approach is presented in Figure 7. The basic primitives of `Format` are redefined to simply emit building elements in a `abstract_document stack`. In a sense, this stack is a primitive document at this point. In order to approximate the basic notions of group, line and text, the stack needs to be post-processed via `eval_abstract_doc` to generate a value of type `document`. `eval_doc` closes the loop: it pretty-prints a document.

This seat-of-the-pants implementation should be refined in the `Format` spirit: we need add a specific type of pretty-printing formatter that would output such a document. True, it sometimes feel like we are trying to artificially fit a square peg in a round hole. Also, the initial languages are not totally equivalent in terms of expressiveness. Even Wadler's and Hughes's approaches have this problem. Yet, bridging the gap in the opposite direction, from combinators to `Format`/Oppen, is not trivial: the main problem is time efficiency. In their works, Chitil and Swierstra [1, 9] arrive at an optimally bounded solution. Actually, Chitil's [1] solution starts from deriving a construction similar to the `abstract_document` of Figure 7.

9. Related work

Pretty-printing with combinators has garnered interest notably in the lazy community. There has been a continuous trend toward, objectively, more efficiency as well as more algebraic considerations, and also, more subjectively, prettier outputs. However it all began with Oppen [6] and work in LISP [11].

[3] `http://gallium.inria.fr/blog/first-release-of-pprint/`

```
type box = Hv of int

type document =
  | Text of string
  | Concat of document * document
  | Group of box * document
  | Line of int

type abstract_document =
  | AText of string
  | AOpenBox of box
  | ABreak of int
  | ACloseBox

let stack = ref []
let push e = stack := e :: !stack
let print_string s = push (AText s)
let print_break n = push (ABreak n)
let open_box b = push (AOpenBox b)
let open_hvbox n = push (AOpenBox (Hv n))
let close_box () = push ACloseBox

let eval_abstract_doc stack =
  let rec loop = function
    | [] -> Text ""
    | AText s :: l -> Concat (Text s, loop l)
    | AOpenBox b :: l ->
      let content, next = accu_until_close l in
      Concat(Group (b, content), loop next)
    | ABreak n :: l -> Concat (Line n, loop l)
    | ACloseBox :: _ -> failwith "No open box to close"
  and accu_until_close  = function
    | [] -> failwith "No closing of open box"
    | ACloseBox :: l -> Text "", l
    | AText s :: l ->
      let content, next = accu_until_close l in
      Concat (Text s, content), next
    | AOpenBox b :: l ->
      let content, next = accu_until_close l in
      let content', next = accu_until_close next in
      Concat(Group (b, content), content'), next
    | ABreak n :: l ->
      let content, next = accu_until_close l in
      Concat(Line n, content), next
  in loop (List.rev stack)

let rec eval_doc ppf = function
  | Text s -> fprintf ppf "%s" s
  | Concat (d1, d2) ->
    fprintf ppf "%a%a" eval_doc d1 eval_doc d2
  | Group (Hv n, d) ->
    fprintf ppf "@[<hv %d>%a@]" n eval_doc d
  | Line n -> pp_print_break ppf n 0
```

Figure 7: Document generation in `Format`

Leijen has implemented Wadler's ideas in Haskell[4]. One drawback of Wadler's proposal is that its complexity is non-linear. This together with the fact that earlier proposals were "not pretty enough", lead Bernardy[5] to offer another Haskell-based library on the same algebraic principles. Chitil [1] offers a very thorough summary of the existing proposals before giving a more efficient solution.

Swierstra and Chitil [9] seem to have the final word, for now, with their combinator-based functional pretty-printing algorithm that retains the linear space and time complexities of Oppen's algorithm. As Oppen's, it does not need to have the full document to start printing out text. This joint article extends both author's previous works [1, 8]. In particular, Chitil [1] has benchmarks showing the relative efficiencies of his proposal, Hughes's and Wadler's. Some years later, Kiselyov, Peyton-Jones and Sabry [4] describe a very elegant use of yield to implement an incremental linear pretty-printer.

Such document-based implementations have also been made for OCaml. For example, Pottier, with `Pprint`, Tayanovsky[6] and Lindig [5] have implemented pretty-printer combinators inspired by Wadler's work. Pottier's module actually implements Leijen's ideas in the OCaml world.

Out of the functional programming communities, there are various implementations of Oppen's algorithm. For example, Giese has implemented Oppen's algorithm in Java[7]. Wadler's algorithm has implementations in languages like Rust[8], or Javascript/Node [9]. To the best of our knowledge, there is no standard library support as in OCaml or no such deep development as in Haskell.

[4] `http://research.microsoft.com/en-us/um/people/daan/download/pprint/pprint.html`
[5] `http://www.cse.chalmers.se/~bernardy/prettiest.html`
[6] `http://t0yv0.blogspot.com/2012/04/prettier-printer-in-ml.html`
[7] `http://jpplib.sourceforge.net/`, forked at `https://io7m.github.io/jpplib/`
[8] `https://github.com/epsilonz/pretty.rs`
[9] `https://github.com/folktale/text.pretty-printing`

10. Conclusion

We have provided an extended introduction to `Format`, its basic and more advanced features, but also replaced it in the more general pretty-printing landscape. Our hope is that it increases the understanding and the use of this module of the OCaml standard library.

Techniques based on Oppen's algorithm do have some limitations since the initial goal was to strike a balance between expressiveness and efficiency. For example, it is impossible for the offset of Oppen boxes to depend on what is going to be printed in the future. `Format` module exhibits the very same limitations. Document-based pretty-printing can have that feature. Indeed, it might have access to the whole document before deciding what to do since the document must be built. In this case, constructing the whole document might render the output prettier.

By and large, we are convinced that `Format` is already a good solution to the problems of pretty-printing data for the working programmer. Yet, it can be improved in a number of ways. We are considering adding fully abstract printing (document production without I/O side effects), printing of polymorphic data structures (not to be confused with now available polymorphic printing of monomorphic values) and tables (column-formatted outputs).

We are currently experimenting with these subjects. We would like to make these extensions work for all modules using format strings (`Printf`, `Scanf`). We are hopeful that it will provide new and interesting ways to handle formatted data in OCaml.

References

[1] O. Chitil. Pretty printing with lazy dequeues. *ACM Trans. Program. Lang. Syst.*, 27(1):163–184, 2005.

[2] C. Dominik. *The Org-Mode 8 Reference Manual: Organize Your Life with GNU Emacs*, 2014.

[3] J. Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *LNCS*, pages 53–96. Springer, 1995.

[4] O. Kiselyov, S. L. P. Jones, and A. Sabry. Lazy v. yield: Incremental, linear pretty-printing. In R. Jhala and A. Igarashi, editors, *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *LNCS*, pages 190–206. Springer, 2012.

[5] C. Lindig. Strictly pretty, 2000.

[6] D. C. Oppen. Prettyprinting. *ACM Trans. Program. Lang. Syst.*, 2(4):465–483, 1980.

[7] R. M. Pirsig. *Zen and the Art of Motorcycle Maintenance.* William Morrow & Company, 1974.

[8] S. D. Swierstra. Linear, online, functional pretty printing (corrected and extended version). Technical Report UU-CS-2004-025a, Institute of Information and Computing Sciences, Utrecht University, 2004.

[9] S. D. Swierstra and O. Chitil. Linear, bounded, functional pretty-printing. *J. Funct. Program.*, 19(1):1–16, 2009.

[10] P. Wadler. A prettier printer. *J. Funct. Program.*, pages 223–244, 1998.

[11] R. C. Waters. User Format Control in a LISP Prettyprinter. *ACM Trans. Program. Lang. Syst.*, 5(4):513–531, 1983.