CIL API Documentation (version 1.3.2)

May 12, 2005

Contents

1 Module `Pretty` : Utility functions for pretty-printing.

The major features provided by this module are

- An `fprintf`-style interface with support for user-defined printers
- The printout is fit to a width by selecting some of the optional newlines
- Constructs for alignment and indentation
- Print ellipsis starting at a certain nesting depth
- Constructs for printing lists and arrays

Pretty-printing occurs in two stages:

- Construct a `Pretty.doc`[1] object that encodes all of the elements to be printed along with alignment specifiers and optional and mandatory newlines
- Format the `Pretty.doc`[1] to a certain width and emit it as a string, to an output stream or pass it to a user-defined function

The formatting algorithm is not optimal but it does a pretty good job while still operating in linear time. The original version was based on a pretty printer by Philip Wadler which turned out to not scale to large jobs.

API

```
type doc
```

The type of unformated documents. Elements of this type can be constructed in two ways. Either with a number of constructor shown below, or using the `Pretty.dprintf`[1] function with a `printf`-like interface. The `Pretty.dprintf`[1] method is slightly slower so we do not use it for large jobs such as the output routines for a compiler. But we use it for small jobs such as logging and error messages.

Constructors for the doc type.

`val nil : doc`

Constructs an empty document

`val (++) : doc -> doc -> doc`

Concatenates two documents. This is an infix operator that associates to the left.

`val text : string -> doc`

A document that prints the given string

`val num : int -> doc`

A document that prints an integer in decimal form

`val real : float -> doc`

A document that prints a real number

`val chr : char -> doc`

A document that prints a character. This is just like `Pretty.text`[1] with a one-character string.

`val line : doc`

A document that consists of a mandatory newline. This is just like `(text "\n")`. The new line will be indented to the current indentation level, unless you use `Pretty.leftflush`[1] right after this.

`val leftflush : doc`

Use after a `Pretty.line`[1] to prevent the indentation. Whatever follows next will be flushed left. Indentation resumes on the next line.

`val break : doc`

A document that consists of either a space or a line break. Also called an optional line break. Such a break will be taken only if necessary to fit the document in a given width. If the break is not taken a space is printed instead.

`val align : doc`

Mark the current column as the current indentation level. Does not print anything. All taken line breaks will align to this column. The previous alignment level is saved on a stack.

`val unalign : doc`

Reverts to the last saved indentation level.

`val mark : doc`

Mark the beginning of a markup section. The width of a markup section is considered 0 for the purpose of computing identation

`val unmark : doc`

The end of a markup section

Syntactic sugar

`val indent : int -> doc -> doc`

Indents the document. Same as `((text " ") ++ align ++ doc ++ unalign)`, with the specified number of spaces.

`val markup : doc -> doc`

Prints a document as markup. The marked document cannot contain line breaks or alignment constructs.

`val seq : sep:doc -> doit:('a -> doc) -> elements:'a list -> doc`

Formats a sequence. `sep` is a separator, `doit` is a function that converts an element to a document.

`val docList : ?sep:doc -> ('a -> doc) -> unit -> 'a list -> doc`

An alternative function for printing a list. The `unit` argument is there to make this function more easily usable with the `Pretty.dprintf`[1] interface. The first argument is a separator, by default a comma.

`val d_list : string -> (unit -> 'a -> doc) -> unit -> 'a list -> doc`

sm: Yet another list printer. This one accepts the same kind of printing function that `Pretty.dprintf`[1] does, and itself works in the dprintf context. Also accepts a string as the separator since that's by far the most common.

```
val docArray : ?sep:doc ->
  (int -> 'a -> doc) -> unit -> 'a array -> doc
```

Formats an array. A separator and a function that prints an array element. The default separator is a comma.

`val docOpt : (unit -> 'a -> doc) -> unit -> 'a option -> doc`

Prints an `'a option` with `None` or `Some`

```
module MakeMapPrinter :
   functor (Map :  Map.S) ->   sig

     val docMap :
       ?sep:Pretty.doc ->
       (Map.key -> 'a -> Pretty.doc) -> unit -> 'a Map.t -> Pretty.doc
```

Format a map, analogous to docList.

```
val d_map :
  ?dmaplet:(Pretty.doc -> Pretty.doc -> Pretty.doc) ->
  string ->
  (unit -> Map.key -> Pretty.doc) ->
  (unit -> 'a -> Pretty.doc) -> unit -> 'a Map.t -> Pretty.doc
```

Format a map, analogous to d_list.

```
end
```

Format maps.

```
val insert : unit -> doc -> doc
```

A function that is useful with the `printf`-like interface

```
val dprintf : ('a, unit, doc) Pervasives.format -> 'a
```

This function provides an alternative method for constructing `doc` objects. The first argument for this function is a format string argument (of type `('a, unit, doc) format`; if you insist on understanding what that means see the module `Printf`). The format string is like that for the `printf` function in C, except that it understands a few more formatting controls, all starting with the @ character.

The following special formatting characters are understood (these do not correspond to arguments of the function):

- @[Inserts an `Pretty.align`[1]. Every format string must have matching `Pretty.align`[1] and `Pretty.unalign`[1].
- @] Inserts an `Pretty.unalign`[1].
- @! Inserts a `Pretty.line`[1]. Just like "\n"
- @? Inserts a `Pretty.break`[1].
- @< Inserts a `Pretty.mark`[1].
- @> Inserts a `Pretty.unmark`[1].
- @^Inserts a `Pretty.leftflush`[1] Should be used immediately after @! or "\n".
- @@ : inserts a @ character

In addition to the usual `printf` % formatting characters the following two new characters are supported:

- %t Corresponds to an argument of type `unit -> doc`. This argument is invoked to produce a document
- %a Corresponds to **two** arguments. The first of type `unit -> 'a -> doc` and the second of type `'a`. (The extra `unit` is do to the peculiarities of the built-in support for format strings in Ocaml. It turns out that it is not a major problem.) Here is an example of how you use this:

```
dprintf "Name=%s, SSN=%7d, Children=@[%a@]\n"
             pers.name pers.ssn (docList (chr ',' ++ break) text)
             pers.children
```

The result of `dprintf` is a `Pretty.doc`[1]. You can format the document and emit it using the functions `Pretty.fprint`[1] and `Pretty.sprint`[1].

`val fprint : Pervasives.out_channel -> width:int -> doc -> unit`

Format the document to the given width and emit it to the given channel

`val sprint : width:int -> doc -> string`

Format the document to the given width and emit it as a string

```
val fprintf :
  Pervasives.out_channel -> ('a, unit, doc) Pervasives.format -> 'a
```

Like `Pretty.dprintf`[1] followed by `Pretty.fprint`[1]

`val printf : ('a, unit, doc) Pervasives.format -> 'a`

Like `Pretty.fprintf`[1] applied to `stdout`

`val eprintf : ('a, unit, doc) Pervasives.format -> 'a`

Like `Pretty.fprintf`[1] applied to `stderr`

`val gprintf : (doc -> doc) -> ('a, unit, doc) Pervasives.format -> 'a`

Like `Pretty.dprintf`[1] but more general. It also takes a function that is invoked on the constructed document but before any formatting is done.

`val withPrintDepth : int -> (unit -> unit) -> unit`

Invokes a thunk, with printDepth temporarily set to the specified value

The following variables can be used to control the operation of the printer

`val printDepth : int Pervasives.ref`

Specifies the nesting depth of the `align`/`unalign` pairs at which everything is replaced with ellipsis

`val printIndent : bool Pervasives.ref`

If false then does not indent

`val fastMode : bool Pervasives.ref`

If set to `true` then optional breaks are taken only when the document has exceeded the given width. This means that the printout will looked more ragged but it will be faster

`val flushOften : bool Pervasives.ref`

If true the it flushes after every print

`val countNewLines : int Pervasives.ref`

Keep a running count of the taken newlines. You can read and write this from the client code if you want

2 Module `Errormsg` : Utility functions for error-reporting

`val logChannel : Pervasives.out_channel Pervasives.ref`

A channel for printing log messages

`val debugFlag : bool Pervasives.ref`

If set then print debugging info

`val verboseFlag : bool Pervasives.ref`

`val warnFlag : bool Pervasives.ref`

Set to true if you want to see all warnings.

`exception Error`

Error reporting functions raise this exception

`val error : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

Prints an error message of the form `Error: ...`. Use in conjunction with s, for example: `E.s (E.error ... )`.

`val bug : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

Similar to `error` except that its output has the form `Bug: ...`

`val unimp : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

Similar to `error` except that its output has the form `Unimplemented: ...`

`val s : Pretty.doc -> 'a`

Stop the execution by raising an Error. Use "s (error "Foo")"

`val hadErrors : bool Pervasives.ref`

This is set whenever one of the above error functions are called. It must be cleared manually

`val warn : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

Like `Errormsg.error`[2] but does not raise the `Errormsg.Error`[2] exception. Use: `ignore (E.warn ...)`

`val warnOpt : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

Like `Errormsg.warn`[2] but optional. Printed only if the `Errormsg.warnFlag`[2] is set

`val log : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

Print something to `logChannel`

`val logg : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

same as `Errormsg.log`[2] but do not wrap lines

`val null : ('a, unit, Pretty.doc) Pervasives.format -> 'a`

Do not actually print (i.e. print to /dev/null)

```
val pushContext : (unit -> Pretty.doc) -> unit
```
Registers a context printing function

```
val popContext : unit -> unit
```
Removes the last registered context printing function

```
val showContext : unit -> unit
```
Show the context stack to stderr

```
val withContext : (unit -> Pretty.doc) -> ('a -> 'b) -> 'a -> 'b
```
To ensure that the context is registered and removed properly, use the function below

```
val newline : unit -> unit
val newHline : unit -> unit
val getPosition : unit -> int * string * int
val getHPosition : unit -> int * string
```
high-level position

```
val setHLine : int -> unit
val setHFile : string -> unit
val setCurrentLine : int -> unit
val setCurrentFile : string -> unit
type location = {
  file : string ;
```
The file name

```
  line : int ;
```
The line number

```
  hfile : string ;
```
The high-level file name, or "" if not present

```
  hline : int ;
```
The high-level line number, or 0 if not present

```
}
```
Type for source-file locations

```
val d_loc : unit -> location -> Pretty.doc
val d_hloc : unit -> location -> Pretty.doc
val getLocation : unit -> location
val parse_error : string -> 'a
val locUnknown : location
```
An unknown location for use when you need one but you don't have one

```
val readingFromStdin : bool Pervasives.ref
```

Records whether the stdin is open for reading the goal *

```
val startParsing : ?useBasename:bool -> string -> Lexing.lexbuf
val startParsingFromString :
  ?file:string -> ?line:int -> string -> Lexing.lexbuf
val finishParsing : unit -> unit
```

3 Module `Clist` : Utilities for managing "concatenable lists" (clists).

We often need to concatenate sequences, and using lists for this purpose is expensive. This module provides routines to manage such lists more efficiently. In this model, we never do cons or append explicitly. Instead we maintain the elements of the list in a special data structure. Routines are provided to convert to/from ordinary lists, and carry out common list operations.

```
type 'a clist =
  | CList of 'a list
```

The only representation for the empty list. Try to use sparingly.

```
  | CConsL of 'a * 'a clist
```

Do not use this a lot because scanning it is not tail recursive

```
  | CConsR of 'a clist * 'a
  | CSeq of 'a clist * 'a clist
```

We concatenate only two of them at this time. Neither is the empty clist. To be sure always use append to make these

The clist datatype. A clist can be an ordinary list, or a clist preceded or followed by an element, or two clists implicitly appended together

```
val toList : 'a clist -> 'a list
```

Convert a clist to an ordinary list

```
val fromList : 'a list -> 'a clist
```

Convert an ordinary list to a clist

```
val single : 'a -> 'a clist
```

Create a clist containing one element

```
val empty : 'a clist
```

The empty clist

```
val append : 'a clist -> 'a clist -> 'a clist
```

Append two clists

```
val checkBeforeAppend : 'a clist -> 'a clist -> bool
```

A useful check to assert before an append. It checks that the two lists are not identically the same (Except if they are both empty)

`val length : 'a clist -> int`

Find the length of a clist

`val map : ('a -> 'b) -> 'a clist -> 'b clist`

Map a function over a clist. Returns another clist

`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b clist -> 'a`

A version of fold_left that works on clists

`val iter : ('a -> unit) -> 'a clist -> unit`

A version of iter that works on clists

`val rev : ('a -> 'a) -> 'a clist -> 'a clist`

Reverse a clist. The first function reverses an element.

```
val docCList :
  Pretty.doc -> ('a -> Pretty.doc) -> unit -> 'a clist -> Pretty.doc
```

A document for printing a clist (similar to `docList`)

4 Module `Stats` : Utilities for maintaining timing statistics

`val reset : bool -> unit`

Resets all the timings. Invoke with "true" if you want to switch to using the hardware performance counters from now on. You get an exception if there are not performance counters available

`exception NoPerfCount`

`val has_performance_counters : unit -> bool`

Check if we have performance counters

`val sample_pentium_perfcount_20 : unit -> int`

Sample the current cycle count, in megacycles.

`val sample_pentium_perfcount_10 : unit -> int`

Sample the current cycle count, in kilocycles.

`val time : string -> ('a -> 'b) -> 'a -> 'b`

Time a function and associate the time with the given string. If some timing information is already associated with that string, then accumulate the times. If this function is invoked within another timed function then you can have a hierarchy of timings

`val repeattime : float -> string -> ('a -> 'b) -> 'a -> 'b`

repeattime is like time but runs the function several times until the total running time is greater or equal to the first argument. The total time is then divided by the number of times the function was run.

`val print : Pervasives.out_channel -> string -> unit`

Print the current stats preceeded by a message

`val lastTime : float Pervasives.ref`

Time a function and set lastTime to the time it took

`val timethis : ('a -> 'b) -> 'a -> 'b`