A (very) short introduction to HDoc

Armin Größlinger

`groessli@fmi.uni-passau.de`

May 2002

Abstract

HDoc generates documentation in HTML format for Haskell modules.[1] The generated documents are cross linked and include summaries and detailed descriptions for the documented functions, data types, type classes and instance declarations.

To put it in other words: HDoc is very much like Javadoc.

Contents

[1]This document describes HDoc 0.8.2.

1 Running HDoc

```
Usage: ../../hdoc [OPTION...] file/module...
  -d DIR            --destdir=DIR      Send output to directory DIR
  -e                --exports          Document exported objects only
                    --line-numbers     Output line numbers
  -t TITLE          --title=TITLE      Set the document title
  -i PATH, -I PATH                     Search PATH for modules
                    --stylesheet=FILE  Set the stylesheet to use
                    --show-symbols     Show all supported named symbols
  -V                --version          Print HDoc's version number
  -h                --help             Print this (short) help
```

The given files are read, parsed and then the HTML documents are constructed (in the current directory, when no `-d DESTDIR` is present).

`files/modules` can be Haskell scripts (`.hs`), literate Haskell scripts (`.lhs`, LaTeX style or "`> ...`" style) or Green Card files (`.gc`)[2]. When a file name does not contain a ".", then it is considered to be a module name and the corresponding module is looked for on the search path (see below).

Per default, HDoc emits documentation for all top level functions, data types, type synonyms, classes and instances. With `--exports` only objects appering in the export list of the modules are included in the output.

When HDoc runs, it will try to load modules imported from the files given on the command line. For example, if you have a file `A.hs` with contents

```
module A where

import B

...
```

then `hdoc A.hs` (or `hdoc A`) will cause HDoc to look for module `B`. It will try to find it in the files `B.gc`, `B.lhs`, `B.hs` in that order. You can use the `-I` or `-i` switch to set the search path for modules (the current directory is always searched), e.g.

```
hdoc -I /the/path/to/moduleB -I ../othermodules A.hs
```

Hierarchical module names are supported, i. e. the module name `X.Y.Z` is translated to the path name `X/Y/Z.{gc,lhs,hs}`.

[2]Support for Green Card files is incomplete.

HDoc will not generate output for modules not listed on the command line, but for things re-exported from these modules. In addition it will use them to derive type signatures for instance declarations etc. HDoc knows about the type classes defined in the standard prelude and in the standard libraries, so the type signatures for instances of Eq, Enum etc. can be calculated by HDoc.

If you don't set a document title with `--title`, HDoc will use a default title.

The appearance of the generated documents is controlled through a stylesheet file. Currently the following format classes are used in documents generated by HDoc[3]:

```
TableHeadingColor
TableRowColor
FrameHeadingFont
FrameItemFont
```

The default stylesheet (named `hdoc.css`) generated by HDoc is

```
body { background: #DDDDDD }
h2 { color: #000000 }

.TableHeadingColor { background: #CCCCFF }
.TableRowColor     { background: #DDDDDD }

.FrameHeadingFont { font-size: normal; font-family: normal }
.FrameItemFont    { font-size: normal; font-family: normal }
```

2 HDoc's input format

HDoc extracts information from two sources:

- it uses HsParser[4] to take type signatures, class/instance declarations, and data type declarations from the Haskell source code.
- information which can't be taken from the source (most importantly the textual descriptions for the documented objects) are taken from special comments.

[3]The class names are the same as in Javadoc; it should therefore be possible to use the same stylesheet for Javadoc and HDoc.

[4]Please note that HsParser currently does not support languge extensions like `foreign import` etc. Therefore HDoc includes a modified version of HsParser which accepts some (but not all) language extensions. See the `KNOWN_PROBLEMS` file in the distribution for details.

2.1 Comment format

The special comments recognized by HDoc exist in two versions:

The new format uses blocks of end-of-line comments which are introduced by `---` for general comments or `--'` for constructor documentation. Examples:

```
---
-- Here is some documentation. The
-- first comments must consist of three dashes,
-- the following lines start with double dashes.
-- Empty lines or code fragments left of the
-- comments are not allowed (i.e. they
-- signal end-of-documentation).

--' This would be documentation for a data type
--  constructor, which can also expand over some lines.
```

Comments in the old format start with `{---` and finish with `-}`. This corresponds to `---`. There is no corresponding old format for `--'`.

You probably want to have a look into the `examples/` directory of the HDoc distribution to get an impression of what comments for HDoc look like and how they are used.

Comments recognized by HDoc consist of three parts:

- Documentation text written in a HTML subset (see section 2.2)
- Tags for specifying more precisely what the comment documents (see sections 2.3 and 2.4)
- Arguments to (some of) the tags, like parameter names or references to other objects

Tags are introduced by a `@` sign. Some of the tags ("general tags", section 2.3) are allowed in every comment, others (section 2.4) are only allowed in comments for specific objects.

2.2 Documentation text

The documentation text is written in a resticted HTML.

Supported Tags:

<a href="…"	…	</a>	reference
<p>	…	</p>	paragraph (note that </p> is required)
<em>	…	</em>	emphasis
<strong>	…	</strong>	stronger emphasis
<code>	…	</code>	font suitable for source code
<pre>	…	</pre>	preformated text
<br>			line break
<b>	…	</b>	bold
<i>	…	</i>	italics
<tt>	…	</tt>	typewriter font

The paragraph tag can have a class="…" attribute to change the appearance of the paragraph.
Supported named symbols (like "&quot;"): run `hdoc --show-symbols` to get a list of recognized named symbols and special characters.

Below we'll use the following symbols:

`DESCRIPTION` HTML text like described above. For functions, data types, type synonyms, classes, and instances the first sentence (the part up to and including the first ".") is used in the short summaries.

`SHORTDESCRIPTION` is like `DESCRIPTION`, but should be short, one sentence or less.

`NAME` is the name of a Haskell identifier, i.e. it must be one word.

`TYPE` denotes a Haskell type or something syntatically similar.

`TYPED` is similar to `TYPE`, but with names (and some types). This allows types and names to be given at the same time. Examples:

```
value :: Integer
(name, age) :: (String, Int)
(name :: String, age :: Int)
(name, age :: Int) :: (String, Int)
Maybe (result :: Double)
list :: [Complex Float]
```

The following special symbols are used in the descriptions below:

`[ ... ]`	optional
`( ... )*`	repeat as often as needed
`( ... | ... )`	alternatives

2.3 General tags

2.3.1 @doc

Comments can appear directly in front of the thing to be documented (applies to `{---... -}` and `---` style comments) or, for `--'` comments, directly after the constructor(s). Alternatively, `{---... -}` and `---` comments can start with `@doc object` and the comment is associated with the source code not by its position, but by the name "`object`" given in the comment. "`object`" can refer to a toplevel function, a function inside a class declaration, a type synonym, a data type or a class. It is currently not possible to name a module, an instance, or a function inside an instance declaration this way (ambiguities would arise).

```
data List a = Nil | Cons a (List a)

f :: [a] -> List a

--- @doc f
--  documentation for f.

--- @doc List
--  documentation for the list data type.
```

2.3.2 @see

`---` and `{---... -}` style comments can have (as their last entries) `@see` tags to refer to other objects, like

```
---
--   ....
-- @see f
-- @see M.D
-- @see (M.+)
```

Targets of `@see` tags can be toplevel functions, functions in class declarations, type synonyms, data types, and classes. The target can be qualified or unqualified and may be put in parenthesis (as in `(M.+)`).

2.4 Tags for specific Haskell objects

2.4.1 Functions

```
---
-- DESCRIPTION
-- ( @param TYPED  [-  SHORTDESCRIPTION] )*
--
-- ( @return | @monadic ) [TYPED]  [-  SHORTDESCRIPTION]
f [ :: TYPE ]
```

This generates documentation for a function `f`. The type signature is automatically derived if `f` belongs to an instance declaration and HDoc can find the module where the corresponding class is defined. In all other cases the type signature must be explicitly given so that HDoc can show it in the output. In contrast to Javadoc the names of the parameters can be chosen arbitrarily, i.e. they need not match the formal parameters of the function definition (this is so because when pattern matching is used then the names for the formal parameters need not be unique, which is not the case for Java).

When no HDoc comment is present, simply the type signature is taken (or calculated in case of an instance) and no further description is produced in the output.

When a `@param` or `@return` / `@monadic` clause does not include a type for the parameter (as for `f` and `result` in the example for `map` below), then HDoc will calculate the type from the type signature.

The difference between @return and @monadic is that

```
---
-- @param  x
-- @return y
f :: Integer -> IO String
```

will produce

```
(y :: IO String) = f (x :: Integer)
```

whereas

```
---
-- @param   x
-- @monadic y
f :: Integer -> IO String
```

will produce

```
(y :: String) <- f (x :: Integer)
```

Note that with `@monadic` HDoc emits `<-` instead of `=` and the type constructor (`IO` in this case) is dropped from the type given for the result. Examples:

```
---
-- Applies a function to every element of a list.
-- @param  f            -  the function to map
--                         over the list.
-- @param  list :: [a]  -  the list of values to apply
--                         <code>f</code> to.
-- @return result       -  the list with <code>f</code>
--                         applied to its values.
map :: (a -> b) -> [a] -> [b]
map f xs = ...

class X a where
  ---
  -- A function in a class.
  -- @param x   -  the argument.
  -- @return y  -  the result.
  f :: a -> a

 instance X Int where
   ---
   -- Implementation of <code>f</code> for
   -- <code>Int</code>s.
   -- @param i   -  the given integer.
   -- @return d  -  the integer doubled.
   f = (*2)
```

These examples will generate documentation for the functions

```
map :: (a -> b) -> [a] -> [b]
f :: X a => a -> a
f :: Int -> Int
```

(and for the class `X a` and its instance `X Int`).

2.4.2 Type synonyms

```
---
-- DESCRIPTION
type TYPE = TYPE
```

No special tags are available for type synonyms currently.

2.4.3 Data types

```
---
-- DESCRIPTION
--
-- ( @cons TYPED  [-  SHORTDESCRIPTION]
-- | @cons NAME { [ TYPED, ]* TYPED } [-  SHORTDESCRIPTION] )*
data TYPE = ...
```

or

```
---
-- DESCRIPTION
data TYPE = ...  --' SHORTDESCRIPTION
       (  | ...  --' SHORTDESCRIPTION )*
```

or the same with `newtype` instead of `data`.

Each `@cons ...` documents a constructor; the second variant is for labelled fields. The special comment `--'` can be used to document one or more constructor(s) after its/their declaration, i.e. in

```
data X = A   --' d1
       | B
       | C   --' d2
       | D   --' d3
       | E
```

`d1` is the documentation for `A`, `d2` is used for `B` *and* `C`, `d3` documents `D`, and `E` does not have any documentation.

You can mix `@cons` with `--'`, but `--'` has precedence over `@cons`.
Examples:

```
---
-- Versatile representation for 2D and 3D points.
-- @cons Point2D  x  y  -  a 2D point with coordinates (x,y).
-- @cons Point3D { x :: a, y :: a, z :: a }   -
--       represents a 3D point with coordinates (x,y,z).
--
data Point a = Point2D a a
             | Point3D { x :: a, y :: a, z :: a }

---
-- Arithmetic expressions.
--
data Expr = Number   Double    --’ a real number
          | Variable String    --’ a variable
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr      --’ arithmetic operations
```

2.4.4 Classes and Instances

```
---
-- DESCRIPTION
class TYPE where
```

or

```
---
-- DESCRIPTION
instance TYPE where
```

This outputs documentation for a class or instance with or without some description. Example:

```
---
-- A class for all kinds of tea.
class Tea a where
    drink :: a -> IO ()
    brew  :: IO a
```

2.4.5 Modules

```
---
-- DESCRIPTION
-- [ @author  SHORTDESCRIPTION ]
```

```
  -- [ @version SHORTDESCRIPTION ]
  module NAME [ EXPORTLIST ] where
```

or

```
  module NAME [ EXPORTLIST ] where
```

Documents a module, of course. Example:

```
  ---
  -- The main module of program XY.
  -- Loads the options, sets up the GUI, makes tea etc.
  -- @author  Jim Hacker
  -- @version 1.0.nearly.perfect
  module Main where

  import GUI
  import MakeTea
```