# Haddock, A Haskell Documentation Tool

Simon Marlow
Microsoft Research Ltd., Cambridge, U.K.

## Abstract

This paper describes Haddock, a tool for automatically generating documentation from Haskell source code. Haddock's unique approach to source code annotations provides a useful separation between the implementation of a library and the interface (and hence also the documentation) of that library, so that as far as possible the documentation annotations in the source code do not affect the programmer's freedom over the structure of the implementation. The internal structure and implementation of Haddock is also discussed.

## Categories and Subject Descriptors

I.7.2 [**Document and text processing**]: Document Preparation—*Languages and systems, Markup languages*

## General Terms

Design, Languages, Algorithms

## Keywords

Haskell, Documentation tool, Documentation generation, Source-code documentation, API documentation, Module system

## 1 Introduction

Generating documentation directly from source code has recently become fashionable, due in no small part to the popularity of Sun's JavaDoc tool[9]. Nowadays most languages have at least one tool for generating documentation from source code [11, 5, 3, 2], and if you program in C or C++ you are in the fortunate position of having a multitude of tools to choose from.

This paper describes Haddock, a documentation tool for Haskell. Figures 1 and 2 give examples of an annotated Haskell module and the corresponding HTML output produced by Haddock, respectively. Haddock improves on other documentation tools in some important ways, as we shall describe later in this section.

Firstly let us be clear about the problem domain: we are primarily interested in generating documentation for a library, or API (application programming interface), rather than generating nicely-formatted source code. In particular *literate programming* systems[6] do not fall into this category; they are concerned with writing well-documented source code, to be later formatted in its entirety. A consumer of an API or library is not interested in the implementation details of the library; indeed, we would rather implementation details were omitted from the documentation wherever possible, for obvious modularity reasons.

There are several compelling reasons to combine library documentation and source code:

- There is less chance that the documentation will stray out of sync with the reality of the implementation, since the documentation is right next to the implementation.
- In most cases there is already documentation in the source code in the form of comments, and there may well be duplication between the comments in the source code and the documentation. Clearly, if the comments can also be interpreted as the documentation itself, then we can eliminate the duplication and furthermore make it easier for the programmer to keep the documentation up to date (and give the programmer an incentive to keep the comments up to date!).
- There is a great deal of documentation that can be extracted automatically from the source code: APIs, types, data structures, class hierarchies, dependency graphs, and so on. Having a tool to extract this information from the actual implementation is more desirable than trying to duplicate it in separate documentation, and even better is a tool that can include programmer-written documentation along with the extracted information.
- Having interpreted the API from the source code, a documentation tool can automatically cross-reference the documentation it produces. If a function type mentions a particular type constructor, for example, it can be hyperlinked or cross-referenced to the definition of that type constructor. The tool can also generate an index from names to definitions, and even an index from names to uses, without intervention from the programmer.

Our documentation tool, Haddock, provides all of these benefits for Haskell source code. In addition, we believe that the following principles are important:

- The form of the documentation annotations we choose to add to the source code should not be restricted to one particular rendering format. For example, it wouldn't do to force the programmer to write documentation annotations in HTML, since that would prevent us rendering the documentation in a medium with less rich formatting facilities. Because we want our annotations to be renderer-independent, we are forced to use a markup format that provides no more that the lowest common denominator of our target rendering formats.
- The programmer spends far more time looking at and editing the source code that he or she does looking at the documentation. Therefore, the source code annotations should be easy to read and write in a plain ASCII editor, without heavyweight markup for common features. Recent discussions on the `haskelldoc` mailing list[1] highlighted this as an important principle for a Haskell documentation system.

So Haddock chooses a lightweight markup format based on that originally used by IDoc[3]. Where possible, documentation markup is simple and mnemonic: for example, we use single-quotes to surround an identifier that should be hyperlinked to its definition.

So far so good. But what makes Haddock different? Well, another good principle for a library-documentation system is this:

- As far as possible, the structure of the implementation of a library should not affect its documentation. Conversely, the desired structure of the documentation should not affect the programmers freedom over the structure of the implementation.

Some concrete examples of this, in the context of Haskell, are:

- A module might define internal functionality which isn't exported to the library consumer; this should not be visible in the documentation either.
- Haskell's module system is flexible in that it allows the internal module structure of a library to be hidden from the library consumer[1]. In Haskell a module may re-export a definition that it has imported from elsewhere; to a consumer of this module this is indistinguishable from a definition which was defined in the module itself.

  Therefore, if the programmer wants to implement his library in multiple modules, but provide a single module which re-exports the external API of the library, then the documentation should mention only the external API.

We will describe how Haddock reconciles these requirements in Section 5.

The final principle that Haddock addresses is this:

- Library documentation often has a structure that is richer than simply a flat list of the functions, types, and classes exported by a module. We often want to separate entities into groups, or further into sections and sub-sections. We might also want to include documentation that is not attached to any particular source-code entity.

[1]With one small exception: internal modules still pollute the module namespace in Haskell 98, because this namespace is flat. There is a proposed extension to hierarchical modules to remedy this.

This, in conjunction with the requirement that the documentation annotations should not impact the structure of the implementation, suggests that the structure of the documentation should not simply follow the order of the definitions in the file, and should be specified independently.

Haddock's primary contribution is that it addresses all of the issues given above, whereas other tools do well on the early principles but tend to fall down on the last two (we compare Haddock with other tools in Section 7). Furthermore, where there is an apparent conflict of interest—the desire for documentation annotations to be next to the source code, and yet to have a separation between the implementation structure and the documentation structure—Haddock finds a useful compromise (see Section 5).

## 2 Overview

Haddock takes a collection of Haskell source modules and produces documentation in one or more output formats. Currently the only fully supported output format is HTML, although there is a partial implementation of a DocBook (SGML) back-end.

The HTML back-end generates the following:

- A root document, which lists all the modules in the documentation (this may be a subset of the modules actually processed, as some of the modules may be hidden; see Section 5.3). If a hierarchical module structure is being used, then indentation is used to show the module structure.
- An HTML page for each module, giving the definitions of each of the entities exported by that module. See Figure 2 for an example.
- A full index for the set of modules, which links each type, class, and function name to each of the modules that exports it.

Haddock understands certain documentation annotations in the Haskell source. Annotations can be used for documenting functions, types or classes, and for adding section headings and other structural cues. The next two sections describe the form of the annotations that Haddock understands.

## 3 Documenting definitions

In this section we describe how Haskell source code can be annotated with documentation for processing by Haddock. Our form of documentation annotations is heavily inspired by IDoc[3].

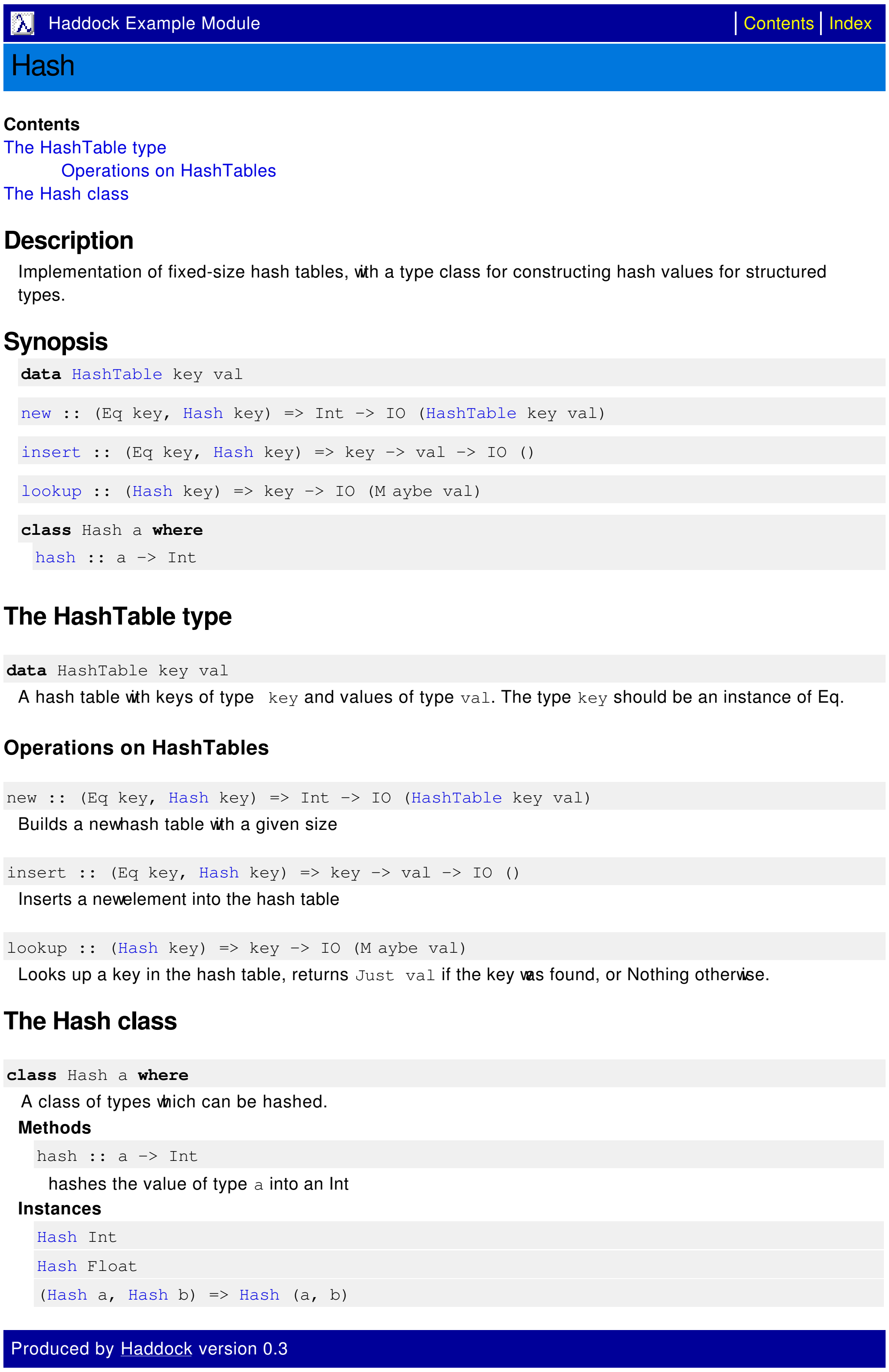
Documentation annotations should of course be ignored by a Haskell compiler, without having to modify each compiler. The traditional way to add annotations to a Haskell source file, in such a way that they will be ignored by a compiler which does not recognise that form of annotation, is to use a *pragma*; indeed, pragmas are even defined by the Haskell 98 standard. If we chose to use a pragma, an annotation might look something like this:

```
{-# DOC This is the documentation for 'f' #-}
f :: Int -> Int
f x = x * x
```

But according to one of the principles given in the introduction, our

```haskell
{- |
  Implementation of fixed-size hash tables, with a type
  class for constructing hash values for structured types.
-}
module Hash (
  -- * The @HashTable@ type
  HashTable,

  -- ** Operations on @HashTable@s
  new, insert, lookup,

  -- * The @Hash@ class
  Hash(..),
 ) where

import Array

-- | A hash table with keys of type @key@ and values of type @val@.
-- The type @key@ should be an instance of 'Eq'.
data HashTable key val = HashTable Int (Array Int [(key,val)])

-- | Builds a new hash table with a given size
new :: (Eq key, Hash key) => Int -> IO (HashTable key val)

-- | Inserts a new element into the hash table
insert :: (Eq key, Hash key) => key -> val -> IO ()

-- | Looks up a key in the hash table, returns @'Just' val@ if the key
-- was found, or 'Nothing' otherwise.
lookup  :: Hash key => key -> IO (Maybe val)

-- | A class of types which can be hashed.
class Hash a where
   -- | hashes the value of type @a@ into an 'Int'
   hash :: a -> Int

instance Hash Int where
   hash = id

instance Hash Float where
   hash = trunc

instance (Hash a, Hash b) => Hash (a,b) where
   hash (a,b) = hash a `xor` hash b
```

**Figure 1. `Hash.hs` (implementations of functions omitted)**



Haddock Example Module | Contents | Index

Hash

**Contents**

The HashTable type

    Operations on HashTables

The Hash class

**Description**

Implementation of fixed-size hash tables, with a type class for constructing hash values for structured types.

**Synopsis**

```
data HashTable key val

new :: (Eq key, Hash key) => Int -> IO (HashTable key val)

insert :: (Eq key, Hash key) => key -> val -> IO ()

lookup :: (Hash key) => key -> IO (Maybe val)

class Hash a where
  hash :: a -> Int
```

**The HashTable type**

```
data HashTable key val
```

A hash table with keys of type `key` and values of type `val`. The type `key` should be an instance of Eq.

**Operations on HashTables**

```
new :: (Eq key, Hash key) => Int -> IO (HashTable key val)
```

Builds a new hash table with a given size

```
insert :: (Eq key, Hash key) => key -> val -> IO ()
```

Inserts a new element into the hash table

```
lookup :: (Hash key) => key -> IO (Maybe val)
```

Looks up a key in the hash table, returns `Just val` if the key was found, or Nothing otherwise.

**The Hash class**

```
class Hash a where
```

A class of types which can be hashed.

**Methods**

```
hash :: a -> Int
```

hashes the value of type `a` into an Int

**Instances**

```
Hash Int
Hash Float
(Hash a, Hash b) => Hash (a, b)
```

Produced by Haddock version 0.3

**Figure 2. Example HTML Output from Haddock (processing `Hash.hs`)**

annotations should be as lightweight as possible, so as to be as easy to read and write as a normal comment. The pragma style is simply too verbose to use for documentation comments.

Instead, we choose to add a single character to the beginning of a comment to indicate a documentation annotation:

```
-- | This is the documentation for 'f'
f :: Int -> Int
f x = x * x
```

The comment form "`-- |`" indicates that what follows is documentation that applies to the following definition[2], which in this case is the type signature for the function `f`. The documentation continues until the first non-comment source line, which is useful if the documentation spans several lines:

```
-- | This is the documentation for 'f',
-- which continues over two lines.
f :: Int -> Int
f x = x * x
```

although in such cases it is sometimes more readable to use the nested form of Haskell comments:

```
{- |
   This is the documentation for 'f',
   which continues over two lines
-}
f :: Int -> Int
f x = x * x
```

If the comment herald is instead "`-- ^`", then it applies to the *previous* definition rather than the following one. Some programmers prefer this style for commenting top-level definitions, but it is also important to be able to document a preceding item when we comment parts of a declaration.

Note that the type signature *must be present* in the source file in order for Haddock to include it in the documentation. Haddock doesn't contain a full Haskell type system (although it does contain a Haskell parser and certain other elements found in a compiler front-end), so it cannot reconstruct omitted type signatures.

## 3.1 Documenting parts of a declaration

Often we want to document not only the declaration as a whole, but also individual parts of it, such as the constructors of a datatype or the arguments of a function.

Haddock allows documentation annotations on parts of a declaration in certain cases. Here is an example of annotations on the constructors of a datatype definition:

```
data T
  = A Int    -- ^ The 'A' constructor
  | B Float  -- ^ The 'B' constructor
```

Note that we use the "`-- ^`" syntax, following the convention that a "`-- ^`" comment documents a *preceding* item. Fields of a record definition can be annotated in a similar way.

[2]The Haskell 98 definition specifies that "`--|`" is a valid token rather than a comment, hence we use the form with the space for Haddock annotations

Annotating methods in a class declaration is just like annotating top-level bindings:

```
class C a where
  -- | A class method 'f'
  f :: a -> Int
```

Finally, function arguments and return values can be documented individually:

```
-- | 'all' tests whether all the elements
-- of a list satisfy a given predicate.
all
  :: (a -> Bool) -- ^ the predicate
  -> [a]         -- ^ the list of elements
  -> Bool        -- ^ returns: 'True', if all the
                 -- elements of the list satisfy the
                 -- predicate, and 'False' otherwise.
```

## 4 Markup

Documentation annotations may include simple formatting and rendering instructions ("markup"). The syntax for the various markup elements is designed to be easy to read and write, and not look overly cluttered when editing the source text in an ASCII editor.

The markup elements understood by Haddock are:

- A Haskell identifier surrounded by single quotes, eg. `'map'`, is rendered in a monospaced font and hyperlinked to its definition, if available.
- A Haskell module name surrounded by double quotes, eg. `"List"` is hyperlinked to the documentation for that module.
- Paragraphs are separated by a blank line.
- Text surrounded by "`@`" symbols is rendered in a monospaced (typewriter) font.
- A string surrounded by angle brackets is interpreted as a URL, and will be hyperlinked if the output format supports it (eg. "`<http://www.haskell.org>`")."
- A paragraph preceded by "`*`" or "`-`" is interpreted as a bulleted paragraph; multiple consecutive bulleted paragraphs become a bulleted list.
- A paragraph preceded by "`(`*n*`)`" or "*n*`.`" where *n* is a number is an numbered paragraph; consecutive numbered paragraphs become an enumerated list (the actual values of *n* are ignored, and paragraphs are numbered starting at 1).
- A paragraph where all the lines begin with ">" is interpreted as a block of code, and rendered in a monospaced font with the ">" symbols removed (but whitespace left intact). This markup style was chosen for consistency with Haskell's existing literate comment style.

## 5 Structuring documentation

One of our goals is to have a rich structure for the documentation generated by Haddock; we would like to be able to structure our documentation into sections, sub-sections and so on, and also include snippets of documentation that are not associated with any particular Haskell entity (a section introduction, for example).

One possible approach which has been adopted by other tools is to include section headings in the source code[3], and to use the order of definitions in the source file as the order in which the entities should be listed in the documentation. However, this isn't ideal for two reasons:

- It links the structure of the documentation with the structure of the implementation, which violates one of the principles we identified in the introduction.
- We don't have a way to group entities that are not defined in the current module, i.e. those that are re-exported from another module. Should these be placed in a section of their own?

Instead, Haddock opts to specify the structure of the documentation independently of the implementation. Fortunately each Haskell module already has a specification of its exports, in the form of an export list; the export list mentions not only the entities defined in the current module but also those that are re-exported, and makes no distinction between the two, which is exactly the property we wish to preserve in the documentation. Furthermore, programmers often use the export list as a way to summarise the exports of a module, by grouping entities into sections and giving type signatures in comments.

Here is an example of a typical export list, which we will annotate:

```
module M (
  -- The type T
  T(..),  -- instance of Eq, Ord, Show

  -- Operations over T
  f, g,

  -- A class C
  C(..),

  -- Operations over C
  j, k
 ) where
```

The exports of the module already fall naturally into sections, all that is needed is to convert the comments into Haddock-specific section annotations:

```
module M (
  -- * The type T
  T(..),  -- instance of Eq, Ord, Show

  -- ** Operations over T
  f, g,

  -- * A class C
  C(..),

  -- ** Operations over C
  j, k
 ) where
```

The comment herald "`-- *`" begins a section heading, where the number of asterisks indicates the section depth: one asterisk is a top-level section, two asterisks is a sub-section, and so on. Haddock will structure the documentation according to the export list, with appropriate section and sub-section headings, and will even provide a contents list at the top of the page (depending on the rendering format).

## 5.1 Named documentation blocks

It is often necessary to include chunks of documentation that don't naturally belong to any particular entity in the module, and don't belong in the module description (the documentation comment at the top of the file). For these situations, Haddock provides two mechanisms:

- A documentation comment can be included inline in the export list, and it will be rendered at the appropriate point in the generated documentation.
- If the documentation is too large to include in the export list without obscuring the structure, it can be given a name, placed in the body of the module, and referred to by name from the export list.

A named block of documentation is introduced using the "`-- $`*name*" form, and referred to using the same form in the export list. For example:

```
module M (
  -- * A section heading
  -- $foo
 )

-- $foo
-- This is a chunk of documentation
-- named $foo.
```

## 5.2 Attributes

Certain attributes can be applied to a module to affect how Haddock produces documentation. The only one we will mention here is the "`hide`" attribute, which specifies that a module should not be included in the generated documentation (more about hidden modules in the next section). Attributes are supplied using the "`-- #`" comment form at the top of the module:

```
-- #hide
module A where
...
```

## 5.3 Re-exports and hiding modules

The Haskell module system provides a way to structure the implementation of a library into multiple modules without exposing that structure to the consumer of the library; this is a powerful abstraction facility. We have seen in the previous section how the structure of the documentation for a module is given by the export list in that module, which means we can include re-exported entities in the structure of our documentation.

Here is a small example of a re-exported datatype. Suppose we have two modules: an implementation-specific module `Internal`, and the module which is to be exposed to the library consumer, `External`:

```
module Internal (T(..)) where
-- | This is the documentation for the 'T' type
data T a = C a

module External (T) where
import Internal
```

Note that even though we are re-exporting `T` from `External`, we still annotate it at the definition site. Haddock will use this documentation, as well as the definition of `T`, when generating the documentation for `External`. The resulting documentation is exactly the same as if we had defined `T` in `External` itself.

What happens when a re-exported entity refers to something from the original defining module? Let's expand the example slightly:

```
module Internal (T(..), f) where
-- | This is the documentation for the 'T' type
data T a = C a
-- | 'f' is a function that operates on an
-- object of type 'T'
f :: T Int -> Int
f (C i) = i * 2

module External (T(..),f) where
import Internal
```

Now the type of `f` refers to `T`. When generating the documentation for `External`, our tool needs to render `f`'s type and documentation, including hyperlinks to the definition of `T`. We have a choice of destination for these hyperlinks: they might point to the definition of `T` exported by `Internal`, or they might point to `T` as exported by `External`. The latter is clearly better, because the implementer of `External` wants to avoid exposing the existence of `Internal` in the documentation altogether. So our implementation has to be careful to re-target hyperlinks when re-exporting entities.

The programmer will probably use the `hide` attribute to declare that `Internal` should not form part of the final documentation (however we still want to process it, because it contains definitions and documentation that we want to propagate to the modules that re-export them). One interesting question is whether in general we can promise not to expose a module such as `Internal` in the documentation. If we stick to strict separate compilation, then it might not always be possible to do so; suppose that we modify the previous example so that the library has two external modules, one which exports the types (`ExtTypes`) and another which exports the operations (`ExtOps`):

```
module ExtTypes (T) where
import Internal

module ExtOps (f) where
import Internal
```

What is interesting about this example is that the module `ExtOps` does not depend, directly or indirectly, on `ExtTypes`. Yet, we want to hyperlink the reference to `T` in the type of `f` to the definition of `T` in the `ExtTypes` interface. If we stick to a policy of strict separate compilation, where each module is processed only after processing its dependents, then this hyperlink is not possible (because the tool will not be aware of the existence of `ExtTypes` when processing `ExtOps`).

Haddock's approach to this problem is to require that `ExtTypes.T` is in scope when processing `ExtOps`, like so:

```
module ExtOps (f) where
import ExtTypes (T)
import Internal
```

This raises another interesting issue: when an entity is in scope via several routes, and we need to hyperlink to it, which instance do we choose? Some instances are better than others:

- We want to avoid linking to entities in *hidden* modules, if possible, so pick a visible instance if there is one.
- If module `A` is a dependent of module `B` (i.e. `B` is further up the module tree), then picking module `B` is usually better on the grounds that `A` is more likely to be an "implementation" module.

In our example above, `T` is available via two routes, `ExtTypes` and `Internal`, but we would choose to link to the version in `ExtTypes` on the grounds both that `Interal` is hidden, and that `ExtTypes` is higher up the dependency tree than `Internal`.

For a given reference, if the only entities available to link the reference to are from hidden modules, then Haddock will emit a warning to the user suggesting that the imports be restructured.

## 6 Implementation

The current implementation of Haddock processes multiple modules simultaneously, and can generate documentation which hyperlinks between the available modules. Referring to entities from modules outside the set of modules being processed is also possible, but for simplicity's sake we will leave that until Section 6.6.

The implementation can be broadly described as following these stages:

- Parse each of the input modules
- Topologically sort the modules into dependency order
- For each module, generate its *interface* (see below)
- Render the set of interfaces in the chosen output format

The first three steps are independent of the last; that is, regardless of which output format we select, the first three stages are identical. Generating output in a different format is a matter of replacing the implementation of the final step. Haddock's current implementation contains two back-ends: HTML, and DocBook[3] (an SGML format designed for technical documentation).

### 6.1 The `Doc` type

Haddock uses a structured data type, `Doc`, to represent user-supplied documentation annotations. Its definition is:

```
data Doc
  = DocEmpty
  | DocAppend Doc Doc
  | DocString String
  | DocParagraph Doc
  | DocIdentifier [HsQName]
  | DocModule String
  | DocEmphasis Doc
  | DocMonospaced Doc
  | DocUnorderedList [Doc]
  | DocOrderedList [Doc]
  | DocCodeBlock Doc
  | DocURL String
```

[3]The DocBook currently back-end lags behind the HTML implementation somewhat in terms of functionality

The syntax is straightforward lightly-marked-up text, with one important addition: it contains embedded identifiers (`HsQName` is a qualified Haskell name), which will be hyperlinked to their definitions in the generated documentation. The `DocIdentifier` constructor contains a *list* of `HsQName` because of overlap in the namespace of Haskell names: a name beginning with an upper-case character can refer to both a class and a constructor, or a type and a constructor. In Haskell source code, the context disambiguates between the two (constructors only occur in expressions, and types/classes occur only in types). In documentation there is no context to tell us which one is meant, so we keep a list of the possible names in the syntax.

There are two important transformations which we will need to perform on a `Doc`:

- *renaming*, which is applying a mapping from `HsQName` to `HsQName` to all the names embedded in the documentation. Recall the design decisions discussed in Section 5.3 which require us to retarget hyperlinks when an entity is re-exported - this is why we need to be able to rename documentation.
- *rendering*, which is mapping `Doc` into the final output format (whatever type that might be).

We facilitate both transformations on `Doc` with a generic mapping function:

```
data DocMap a = DocMap {
  docEmpty         :: a,
  docString        :: String -> a,
  docParagraph     :: a -> a,
  docAppend        :: a -> a -> a,
  docIdentifier    :: [HsQName] -> a,
  docModule        :: String -> a,
  docEmphasis      :: a -> a,
  docMonospaced    :: a -> a,
  docUnorderedList :: [a] -> a,
  docOrderedList   :: [a] -> a,
  docCodeBlock     :: a -> a,
  docURL           :: String -> a
  }

mapDoc :: DocMap a -> Doc -> a
mapDoc m DocEmpty
  = docEmpty m
mapDoc m (DocAppend d1 d2)
  = docAppend m (mapDoc m d1) (mapDoc m d2)
...
```

Now we can implement renaming as an instance of `DocMap`:

```
type NameEnv = FiniteMap HsQName HsQName

mapIdent :: NameEnv -> DocMap Doc
mapIdent fm = DocMap {
  docEmpty      = DocEmpty,
  docAppend     = DocAppend,
  docIdentifier = lookupId fm
  ...
  }

lookupId :: NameEnv -> [HsQName] -> Doc
lookupId fm ns =
  case (catMaybes (map lookupFM ns)) of
    []  -> DocString (show (head ns))
    ns' -> DocIdentifier ns'
```

Note that when we apply the renaming mapping, if none of the required names are present in the mapping then we leave a `String` representing the original name in place in the documentation.

Haddock contains a small lexer and parser to convert `String`s into `Doc` during parsing; as each documentation comment is read, it is parsed into `Doc` (which can elicit a parse error - mismatched quotes, for example). The parser is implemented using Happy[7].

## 6.2 Parsing

Haddock's implementation is based on a freely available generic Haskell parser distributed with GHC[10], which is implemented in Happy. Unfortunately we couldn't make use of the parser as is, since we needed to augment the grammar with extra productions to handle documentation, and extend the abstract syntax to include documentation annotations. The Haddock implementation therefore contains a modified version of the original generic parser.

The modifications we made are described in the following three sections.

### *6.2.1 Lexical analyser*

The `Token` datatype used by the lexical analyser was extended with new tokens for documentation annotations, five in all: documentation annotations (`-- |`, `-- ^`), section headings (`-- *`), named documentation (`-- $`), and options (`-- #`). The lexical analyser was modified to interpret these types of comments (and their nested equivalents) and return the appropriate tokens.

The documentation inside the annotation is parsed into `Doc` as it is read by the lexer; this was done to avoid having to store both `String` and `Doc` in the abstract syntax which would require either abstracting the abstract syntax over the type of documentation, or using an ugly `Either` type.

### *6.2.2 Abstract syntax*

The Haskell abstract syntax datatype was augmented in several places to include documentation annotations. Two new declarations were added (for `-- |` and `-- ^`), and extra fields for documentation were added to constructor declarations, record field declarations, types (to support annotating type arguments), and modules. Also, documentation annotations were added to the export list syntax.

One other change was made to the abstract syntax, which wasn't strictly necessary but simplified the implementation of Haddock. Names in the syntax were previously represented by `Strings`[4], but in Haskell a name can have separate meanings depending on the namespace: a name beginning with an upper-case character can be both a data constructor and a type constructor. Since the implementation of Haddock makes extensive use of mappings whose domain is names (e.g. during renaming of source code), the overlapping namespaces would require us to maintain two separate mappings. However, if a name contains its namespace (trivial to add during parsing because we know from the context which namespace a name belongs to), then we can keep a single mapping. This simplifies the higher-level implementation at the expense of an extra

[4]more or less

level of constructor inside a name, but we found the change to be worthwhile. Our definition of HsQName is the following:

```
data HsQName = Qual Module HsName
             | UnQual HsName

data HsName  = HsTyClsName HsIdentifier
             | HsVarName HsIdentifier
```

An HsIdentifer can be thought of as a String. The HsName type separates the two namespaces: an HsTyClsName is a type or class name, and an HsVarName is a variable or constructor name.

### *6.2.3 Grammar*

The Haskell grammar was extended to include documentation annotations. One interesting thing to note here is that Haddock uses layout to disambiguate annotations; for example

```
class C a where
  f :: a -> a
  -- ^ documentation for 'f'
```

without the layout cues, there's no telling whether the annotation belongs to the function f or the class C. So Haddock uses layout in the same way as a Haskell compiler; a documentation annotation in a declaration list is treated as an individual declaration, and must be separated from other declarations by a semi-colon. The layout system in the parser does this automatically when implicit layout is being used, but if the programmer uses explicit layout, there will be some extra semi-colons to add:

```
class C a where {
  -- | documentation for 'f'
  ;
  f :: a -> a
 }
```

Note the semi-colon is on a separate line so it doesn't get swallowed up by the comment! (Using a nested-style comment would avoid that problem). The extra semi-colon won't confuse a compiler, because empty declarations are allowed in Haskell 98.

## 6.3 Interfaces

Our intention is eventually to resolve names in type signatures and other types of declarations to point to the definitions of the entities they refer to, so that we can construct a hyperlinked rendering of the type in the documentation where each type name is linked to its definition.

Firstly, let's introduce some terminology:

**Name** A name is an unqualified name, of type HsName.

**Source name** A source name is a qualified or unqualified name as it appears in a Haskell source file. In the abstract syntax, it has type HsQName. A source name only has meaning within the scope of a particular module.

**Export name** An export name is represented by an HsQName which must be qualified. The export name M.x refers to the x exported by module M (however M might not be the module that actually *defines* x). An export name has a meaning independent of any particular module.

**Original name** An original name is a qualified name, which points to the *original defining module* of the declaration with that name. We also represent original names using HsQName (to avoid having to parameterise the abstract syntax), but there is a convention that they must be instances of the Qual constructor.

Every entity (function, type, class, constructor, etc.) has a unique original name. It may have several export names, depending on which module(s) export it. Within a given module, it may have zero or more source names. Most qualified source names are also export names, the exceptions being those names imported using the syntax import A as B. Here's an example:

```
module A (x) where
x :: Int

module B (C.x) where
import A
import A as C
```

So the entity with original name A.x has export names A.x and B.x. Within module A it has the source name x and A.x, and within module B it has the source names x, A.x and C.x.

Most Haskell front ends are concerned with only source names and original names, and the act of resolution consists of mapping one to the other. However, in Haddock, we don't always want our hyperlinks to point to the original defining module of an entity (because that module might be internal or hidden), so we introduce the concept of an export name. To state our goal more precisely then: we want to resolve all names in the source code to one of the possible export names for the entity that the name refers to.

In order to resolve names, we must be able to build a mapping from source names to export names. We therefore have to know what names are in scope at the top level of a given module, and which entities they refer to. To do this, we have to know what names and entities are exported by each of the modules imported into the current module. Therefore, we have to process modules in topological order, starting with the modules which have no dependents. The next stage therefore is to topologically sort the modules into dependency order, using a directed graph constructed from the imports of each module.

Next, for each module we construct an *interface*, which collects together all the information we need to know about the module in order to (a) render the documentation for that module, and (b) to build the interfaces for any modules which depend on it.

An Interface is a record type:

```
data Interface = Interface {
    iface_env          :: FiniteMap HsName HsQName,
    iface_sub          :: FiniteMap HsName [HsName],
    iface_decls        :: FiniteMap HsName HsDecl,
    iface_insts        :: [HsDecl],
    iface_orig_exports :: [ExportItem],

    iface_doc          :: Maybe Doc,
    iface_options      :: [DocOption],
    iface_exports      :: [ExportItem]
  }
```

The first five fields, iface_env, iface_sub, iface_decls, iface_insts, and iface_orig_exports, are used when constructing the interfaces for modules that depend on the current module. The other three fields, iface_doc, iface_options, and iface_exports, are used to render the documentation for this module (more about these later).

The first field, iface_env, is a mapping from names to original names which expresses the names exported by this module and the original entities they refer to. This mapping contains *all* the names exported by the module, regardless of whether they were explicitly named in the export list or not, including constructors, record field names, class methods, and so on.

Why do we bother with original names, instead of providing a mapping from source names to export names? Well, when trying to determine which is the "best" export name to use for a particular source name, we must be able to compare different imported names to determine whether they refer to the same entity. An original name has the useful property that it is unique; if we want to compare two names to determine whether they refer to the same entity, we can first find their original names and then compare those.

The second field, iface_sub, maps each name defined in this module to its list of *subordinates*, where a name *a* is a subordinate of a name *b* if *a* can be placed in parenthesis after *b* in an export list. The subordinate name mapping is used to give meaning to import specifiers such as T(..).

The iface_decls and iface_insts field of an interface give the declarations and instances defined in this module respectively. The iface_decls field is restricted to those declarations which are actually exported.

## 6.4 Export Items

The iface_exports field of an interface gives the documentation for a module, expressed in terms of ExportItems, which are defined as follows:

```
data ExportItem
  = ExportDecl    HsDecl
  | ExportSection Int Doc
  | ExportDoc     Doc
  | ExportModule  Module
```

A module's documentation is a list of ExportItems. Each item represents something that should be placed in the documentation:

**ExportDecl** This is a declaration that should be rendered in the documentation. It has been trimmed to remove any parts that are not visible in the exported interface (eg. constructors are removed from an abstract datatype). The declaration is one of the following: a type signature, or a **type**, **data**, **newtype**, or **class** declaration. That is, we don't render any expressions in the documentation, only types.

**ExportSection** A section heading, including its level (section, subsection, etc.),and a Doc representing the text of the heading itself.

**ExportDoc** Some unattached documentation, perhaps included directly from the export list, or maybe included as a result of a reference to a named documentation object.

**ExportModule** A reference to another module. These arise as the result of a complete re-export of a module (i.e. using the form module M in the export list), where the module in question is exported in its entirety and isn't hidden.

For the list of ExportItems in the iface_exports field of an interface, any names embedded in the declarations are *export names*. In the list of ExportItems in the iface_orig_exports field of an interface, the names are all original names. The reason for having two different versions of the ExportItem list will become apparent in the next section.

## 6.5 Algorithm for constructing the interface

The high-level algorithm for constructing an Interface from the parsed Haskell source module is as follows:

1. iface_doc and iface_options can be extracted directly from the abstract syntax.
2. Traverse the top-level declarations, attaching documentation annotations to the declarations they refer to.
3. Build a mapping from source names to original names for the current module, using the interfaces for any imported modules. The subset of this mapping that maps the names exported by this module to original names becomes the value of iface_env for this module. Note: doing this properly requires implementing all the dark corners of the Haskell module system[4].
4. Rename the source code of the module so that each source name is replaced by an original name. The resulting declarations can be used to construct the values of iface_decls and iface_insts for this module's interface.
5. Build a list of ExportItems by traversing the export list:
   - Each entity exported is replaced by an ExportDecl containing the declaration it refers to. The declaration contains original names, and is pruned so that any parts of the declaration which are not visible according to the export are removed.

     If there is no declaration for the entity (perhaps because it is a function without a type signature), then ignore the export and emit a warning to the user.
   - Each export of the form module M is replaced by either ExportModule M, if M is visible and re-exported in its entirety, or the appropriate subset of the contents of iface_orig_exports from M's interface, otherwise.
   - Each section heading is replaced by an ExportSection.
   - Documentation annotations and named documentation in the export list give rise to an ExportDoc.

   The result of this step is the value of iface_orig_exports for the current module.
6. Build a mapping from original names to preferred export names. As we described in Section 5.3, an export name is to be preferred if it
   - refers to a visible module, and
   - occurs later in the topological sort of the module dependency graph.
7. Rename the ExportItems constructed in step 5 using the mapping constructed in step 6, to yield the value of iface_exports for this module.

Having constructed an `Interface` for each module, we can now render the documentation in the format of our choice.

## 6.6 Persistent interfaces

It is important that documentation for a new library can refer to (or hyperlink to) the documentation for the libraries on which it depends. This is especially important when we consider that one of the libraries on which virtually everything depends is the Haskell Prelude. The documentation for these existing libraries may be either in local storage or on the web somewhere; either way we would like to be able to generate documentation which links to it correctly.

The approach we take is to create a *persistent interface* whenever Haddock generates documentation for a set of modules. The persistent interface contains, for each module, a cut-down representation of the `Interface` type described in Section 6.3; essentially we just store the `iface_env` and `iface_sub` components.

When generating documentation for a new set of modules, the user can specify a list of persistent interfaces to read. For each interface, the location of the documentation for that interface is specified (the documentation is assumed to be in the same format as the documentation being generated). Haddock then has available the list of original names exported by each module described by those interfaces, and can generate accurate hyperlinks from the documentation being generated to the existing documentation for other modules referred to.

The idea is that when a library is installed on a user's machine, it should come with a Haddock persistent interface which can be used to generate documentation for modules which use the library. The actual documentation may or may not be local; the same interface file works for both. The process can be made even easier using GHC's package system: in GHC, libraries are grouped into *packages*, where each package has a set of characteristics defined in a configuration file. The characteristics for a package consist of things like directory locations for the interface files for that library, the location of the binary library to be linked in, extra include files and compiler options to use when using that library, and so on. Our plan is to also store the location of the Haddock interface and local documentation (if installed) in the package description, and have Haddock understand packages in the same way as GHC.

Note that because the persistent interface only contains the exported names, and not the full export items, it isn't possible to re-export an entity from a module in another library. This decision was made for purely practical reasons: if we stored full type signatures and documentation in the persistent interface, these files would become huge and slow to read.

## 6.7 Implementation notes

In this section we collect meta-information about the implementation which might be of interest.

The following additional libraries were used in the construction of Haddock:

**HsParser** The generic Haskell parsing library; as previously mentioned this library had to be modified for Haddock.

**HTML** An HTML combinator library (almost a domain-specific language, in fact) for generating HTML. Again, we didn't use the stock version, but modified it slightly to fix bugs and bring it up to date with the latest version of HTML.

**FiniteMap** (distributed with GHC) A rather good balanced-binary-tree implementation of finite maps.

**Regex** (distributed with GHC) An interface to the C regular expression library, used for various small parsing jobs in Haddock.

**Digraph** (from the GHC sources) A good implementation of directed graphs, including linear-time implementations of topological sort and strongly-connected components.

The whole implementation of Haddock, excluding comments and the above libraries, is roughly 2200 lines of Haskell. Broken down into the various parts:

- Types & Miscellaneous: 350 lines
- Front-end
  - Parser & lexer for documentation strings: 180 lines
  - Renaming: 180 lines
  - Computing interfaces: 450 lines
- Back-end
  - HTML renderer: 900 lines
  - DocBook renderer: 130 lines

Interestingly, the back-end renderer for HTML contains almost as much code as the entire front-end. Although there is nothing particularly interesting about the implementation of the back end, it consumed more than its fair share of development time.

## 7 Comparison with other systems

In this section we compare Haddock to the competition. The comparison is by no means complete; there are a plethora of documentation generation systems out there, particularly for C and C++, but since they all have similar functionality we choose a few representative candidates to compare against.

JavaDoc[9] is Sun Microsystems' Tool for documenting Java code, designed specifically for generating HTML. JavaDoc is the forerunner of many of the other documentation-generation tools for other languages; it is a mature tool in widespread use by the Java community. It allows documentation to be included in comments in the Java source code, where the form of the documentation is a mixture of text marked up in HTML and JavaDoc-specific tags. Compared to Haddock, it doesn't allow free-form structuring of the documentation, and the markup syntax is rather verbose.

Java does not allow transparent re-exporting in the same way as the Haskell module system, but it does have inheritance and overriding. JavaDoc handles inheritance and overriding by propagating documentation from the original method to the overriding method where necessary.

HDoc[5] is JavaDoc for Haskell — the comment form is similar, as is the generated documentation. Compared to Haddock, it has fixed-form document structuring, and re-exports are not transparent (they are listed in a separate section in the generated documentation).

IDoc[3] is described as a "no frills" Haskell documentation system; it also generates documentation from Haskell source, but without parsing the source. Hence it has a simpler implementation than Haddock and HDoc, but some of the work has to be done by the programmer: for example the programmer must specify which functions are exported (by annotating them), and which datatypes are to be exported abstractly. IDoc doesn't cope with re-exports, and it doesn't add hyperlinks to the documentation.

OCamldoc[2] is JavaDoc for O'Caml. The comment form is again similar to JavaDoc, although the markup syntax is specific to the OCamldoc tool rather than being plain HTML.

Doxygen[11] is a well-established documentation tool originally for C and C++, but also with support for Java and IDL. Because C and C++ have weak module systems, Doxygen contains its own support for grouping definitions into "modules" and "groups", using special directives in the documentation annotations. In this way, Doxygen supports a separation between the documentation structure and the implementation structure, but the notation is rather more clumsy than Haddock's structured export lists. Doxygen also avoids the problem of having to figure out which copy of a definition to link to, by restricting definitions to reside within a single group only.

## 8 Conclusion and further work

Haddock fulfills the design criteria outlined in the introduction: it generates documentation from source code, with documentation annotations that are easy to read and write, doesn't restrict or expose the implementation details of the library, and it allows a free-form structuring of the final documentation.

Some avenues for further work are outlined in the following sections.

### 8.1 Documentation in the interpreter

Two places which would benefit from automatically-generated documentation is an interpreter or development environment - having documentation for an identifier available from a single mouse click or a single command at the interpreter's prompt would be extremely useful.

If the package system were extended to include information about Haddock documentation as suggested in Section 6.6, then the GHCi interpreter would have immediate access to the documentation for every library available to it, and could pop up the documentation for any type, class, or function on demand.

### 8.2 Type Searching

Haddock has access to a wealth of information about the source code it is processing, and there are a number of useful analyses that can be performed given this information. One idea is to answer questions such as "tell me all the functions that mention type `T`". This can be taken a step further, for instance allowing functions to be searched based on a type *pattern* ("tell me all the functions that take an `A` and return a `B`, regardless of what other arguments they take"). Rittri's work[8] shows that matching types modulo isomorphism is a useful idiom for library searching.

## 9 References

[1] The HaskellDoc mailing list. `http://www.haskell.org/mailman/listinfo/haskelldoc/`.

[2] OCamldoc. `http://caml.inria.fr/devtools/ocamldoc/ocamldoc.html`.

[3] M. Chakravarty. IDoc - a no frills haskell interface documentation system. `http://www.cse.unsw.edu.au/~chak/haskell/idoc/`.

[4] I. S. Diatchki, M. P. Jones, and T. Hallgren. A formal specification of the Haskell 98 module system. In *Haskell Workshop*, Pittsburgh, Pennsylvania, October 2002. ACM.

[5] A. Groesslinger. HDoc. `http://www.fmi.uni-passau.de/~groessli/hdoc/`.

[6] D. E. Knuth. Literate programming. Technical Report STAN-CS-83-981, Department of Computer Science, Stanford University, 1983.

[7] S. Marlow. Happy, a parser-generator for haskell. `http://www.haskell.org/happy/`.

[8] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, 1991.

[9] Sun Microsoystems. Javadoc. `http://java.sun.com/j2se/javadoc/`.

[10] The GHC Team. The Glasgow Haskell Compiler (GHC). `http://www.haskell.org/ghc/`.

[11] D. van Heesch. Doxygen. `http://www.doxygen.org/`.